STIC Search Report
EIC 2100

STIC Database Tracking Number 213048

To: Cheryl Lewis
Location: Patent Training Academy
Art Unit: 2167
Thursday, January 18, 2007

Case Serial Number: 10/791586

From: Alyson Dill
Location: EIC 2100
Randolph 4-b-28
Phone: 272-3527

Alyson.Dill@uspto.gov

Search Notes

A fast and focused search was executed on CDB NPL abstract files. An inventor search and was not executed per our discussion. Please let me know if you require any additional information or require a refocus of the search to retrieve references on an additional aspect of the search.

A selected number of articles were retrieved and printed out in full text.

Thanks.

Alyson Dill
2-3527

SYSTEM:OS - DIALOG OneSearch
File 8:Ei Compendex(R) 1970-2007/Jan W1
(c) 2007 Elsevier Eng. Info. Inc.
*File 8: The file has been reprocessed and accession numbers have changed. See HELP NEWS988 for details.
File 35:Dissertation Abs Online 1861-2006/Nov
(c) 2006 ProQuest Info&Learning
File 65:Inside Conferences 1993-2007/Jan 17
(c) 2007 BLDSC all rts. reserv.
File 2:INSPEC 1898-2007/Dec W4
(c) 2007 Institution of Electrical Engineers
*File 2: UD200612W3 is the last update for 2006. UD200701W1 will be the next update. The file is complete.
File 94:JICST-EPlus 1985-2007/Jan W2
(c)2007 Japan Science and Tech Corp(JST)
*File 94: UD200609W2 is the last update for 2006. UD200701W1 is the first update for 2007. The file is complete and up to date.
File 6:NTIS 1964-2007/Jan W2
(c) 2007 NTIS, Intl Cpyrght All Rights Res
File 144:Pascal 1973-2007/Dec W2
(c) 2007 INIST/CNRS
File 34:SciSearch(R) Cited Ref Sci 1990-2007/Jan W1
(c) 2007 The Thomson Corp
File 434:SciSearch(R) Cited Ref Sci 1974-1989/Dec
(c) 2006 The Thomson Corp
File 99:Wilson Appl. Sci & Tech Abs 1983-2007/Dec
(c) 2007 The HW Wilson Co.
File 266:FEDRIP 2006/Dec
Comp & dist by NTIS, Intl Copyright All Rights Res
File 95:TEME-Technology & Management 1989-2007/Jan W2
(c) 2007 FIZ TECHNIK
File 583:Gale Group Globalbase(TM) 1986-2002/Dec 13
(c) 2002 The Gale Group
*File 583: This file is no longer updating as of 12-13-2002.
File 256:TecInfoSource 82-2007/Aug

(c) 2007 Info.Sources Inc
File 56:Computer and Information Systems Abstracts 1966-2006/Dec
(c) 2006 CSA.
File 60:ANTE: Abstracts in New Tech & Engineer 1966-2006/Dec
(c) 2006 CSA.

Set Items Description
--- ----- -----------

? ds

Set Items Description
S1 2655549 INSTALL?????? ? OR REINSTALL?????? ? OR LOAD??? ? OR PUSH-
??? ?
S2 5229514 SOFTWARE OR FILE OR APPLICATION OR DRIVER? ?
S3 628029 DEPLOY???? ? OR UPGRAD???? ? OR UPDAT?
S4 231563 NAME OR NAMES OR NAMING
S5 4426 (OVER () WRIT??? ?) OR OVERWRIT?
S6 0 1 AND 2 AND 3 AND 4 AND 5
S7 99 (1 OR 3) AND 2 AND 5
S8 81 S7 AND PY<2004
S9 10234067 PREVENT? OR PROHIBIT? OR STOP? OR DETER? OR BARS OR BARR-
ED OR BARRING OR BLOCK OR BLOCKS OR BLOCKED OR BLOCKING
S10 22 4 AND 5
S11 2 (1 OR 3) AND 10
S12 100 7 OR 11
S13 82 S12 AND PY<2004
S14 60 RD (unique items)
S15 106247 DRIVER
S16 1 14 AND 15
S17 1440309 DEVICE OR DRIVER
S18 5 S7 AND S17
S19 5 RD (unique items)
S20 4 S19 NOT PY>2003

S21 4 S20 OR S16
S22 57 S14 NOT S21
S23 13 S17 AND (S1 OR S3) AND S5
S24 13 S23 NOT S22
S25 8 S24 NOT PY>2003
S26 7 RD (unique items)
S27 316 DRIVER (5N) PACKAGE
S28 2251 INSTALLER
S29 261 DRIVER (5N) FILE?
S30 0 27 AND 28 AND 29
S31 0 27 AND 28
S32 0 28 AND (27 OR 29)
S33 773343 INSTALL?
S34 12 S27 AND S33
S35 0 5 AND 34
S36 7 DRIVER (5N) PACKAGE? (10N) (FILE OR FILES)
S37 6 RD (unique items)
S38 6 S37 NOT PY>2004
S39 6 S38 NOT (S34 OR S26 OR S22 OR S13)

14/5/1 (Item 1 from file: 8)
DIALOG(R)File 8:Ei Compendex(R)
(c) 2007 Elsevier Eng. Info. Inc. All rts. reserv.

09865656 E.I. No: EIP04228180503
Title: Achieving database accountability and traceability using the bitemporal relation
Author: Chen, Hsiang-Hui; Farn, Kwo-Jean; Tsai, Dwen-Ren
Conference Title: Proceedings: 37th Annual 2003 International Carnahan Conference on Security Technology
Conference Location: Taipei, Taiwan Conference Date: 20031014-20031016
Sponsor: IEEE Lexington Section; IEEE Aerospace and Electronic System Society; Chung Shan Institute of Science and Technology; U.S. Army Research Laboratory Far East Research Office; et al
E.I. Conference No.: 62905
Source: IEEE Annual International Carnahan Conference on Security Technology, Proceedings 2003. (IEEE cat n 03CH37458)

Publication Year: 2003
CODEN: 85QRAQ
Language: English
Document Type: CA; (Conference Article) Treatment: T; (Theoretical)
Journal Announcement: 0406W1

Abstract: Database systems have become the most crucial constructing components of data stores underlying modem **application** systems. Popular role-based access control model by Sandhu R.S. and E.J. Coyne proposed a way to manage users' access rights. However, employees playing several roles sometimes acquire access rights above their duties. These employees, for their own benefits, are capable of accessing data illegally, modifying or inserting data temporary then illegally outputting data, and finally changing data back to their original status that satisfy integrity of database contents. Usually databases are **updated** through record **overwriting** or deleting and are difficult to trace each user transaction. Hence, owners of these database systems, potentially, might become victims of data temporary misuse by criminals Ooi, Goh, and Tan proposed a dimension space transformation concept based on indexing bitemporal databases1998, which states the concept of transforming one-dimensional time domain to two-dimensional x-y coordinates. In this paper, we first study state-of-the-art of access control methods, then address role conflicts in access rights, and finally discuss the Bitemporal Relation with valid and process time attributes. We further propose an approach, recording database usage trails and transparent to general users, to accomplish all record queries and changes, including insertion, deletion, modification, and retrieving, referenced Ooi, Goh, and Tan's work. Hopefully, this approach might facilitate achieving forensic objects of database traceability and accountability. 11 Refs.

Descriptors: *Database systems; Security of data; Data acquisition; Personnel; Control systems; Societies and institutions; Mathematical models
Identifiers: Accountability; Bitemporal relation; Computer audit; Forensic; Role-based access control model; Traceability
Classification Codes:
901.1.1 (Societies & Institutions)
723.3 (Database Systems); 723.2 (Data Processing); 912.4 (Personnel); 731.1 (Control Systems); 901.1 (Engineering Professional Aspects)
723 (Computer Software, Data Handling & Applications); 912 (Industrial

Engineering & Management); 731 (Automatic Control Principles & Applications); 901 (Engineering Profession); 921 (Applied Mathematics)
72 (COMPUTERS & DATA PROCESSING); 91 (ENGINEERING MANAGEMENT); 73
(CONTROL ENGINEERING); 90 (ENGINEERING, GENERAL); 92 (ENGINEERING
MATHEMATICS)

14/5/3 (Item 3 from file: 8)
DIALOG(R)File 8:Ei Compendex(R)
(c) 2007 Elsevier Eng. Info. Inc. All rts. reserv.

09195315 E.I. No: EIP02467196317
Title: Unroll-based copy elimination for enhanced pipeline scheduling
Author: Kim, Suhyun; Moon, Soo-Mook; Park, Jinpyo; Ebcioglu, Kemal
Corporate Source: School of Elec. Eng. and Comp. Sci. Seoul National University, Seoul 151-742, South Korea
Source: IEEE Transactions on Computers v 51 n 9 September 2002. p 977-994
Publication Year: 2002
CODEN: ITCOB4 ISSN: 0018-9340
Language: English
Document Type: JA; (Journal Article) Treatment: T; (Theoretical)
Journal Announcement: 0211W3
Abstract: Enhanced pipeline scheduling (EPS) is a **software** pipelining technique which can achieve a variable initiation interval (II) for loops with control flow via its code motion pipelining. EPS, however, leaves behind many renaming copy instructions that cannot be coalesced due to interferences. These copies take resources and, more seriously, they may cause a stall if they rename a multilatency instruction whose latency is longer than the II aimed for by EPS. This paper proposes a code transformation technique based on loop unrolling which makes those copies coalescible. Two unique features of the technique are its method of determining the precise unroll amount, based on an idea of extended five ranges, and its insertion of special bookkeeping copies at loop exits. The proposed technique enables EPS to avoid a serious slowdown from latency handling and resource pressure, while keeping its variable II and other advantages. In fact, renaming through copies, followed by unroll-based copy elimination, is EPS's solution to the cross-iteration register **overwrite** problem in **software** pipelining. It works for loops with arbitrary control flow that EPS must deal with, as well as for straightline loops. Our empirical study performed on a VLIW testbed with a two-cycle **load** latency shows that 86 percent of the otherwise uncoalescible copies in innermost loops become coalescible when unrolled 2.2 times on average. In addition, it is demonstrated that the unroll amount obtained is precise and the most efficient. The unrolled version of

the VLIW code includes fewer no-op VLIWs caused by stalls, improving the performance by a geometric mean of 18 percent on a 16-ALU machine. 20 Refs.

Descriptors: *Pipeline processing systems; Computer **software**; Very long instruction word architecture; Scheduling; Codes (symbols); Iterative methods

Identifiers: Pipeline scheduling

Classification Codes:

722.4 (Digital Computers & Systems); 723.2 (Data Processing); 921.6 (Numerical Methods)

722 (Computer Hardware); 723 (Computer Software, Data Handling & Applications); 921 (Applied Mathematics)

72 (COMPUTERS & DATA PROCESSING); 92 (ENGINEERING MATHEMATICS)

21/5/1 (Item 1 from file: 35)
DIALOG(R)File 35:Dissertation Abs Online
(c) 2006 ProQuest Info&Learning. All rts. reserv.

01895964 ORDER NO: AADAA-I3056109
Evaluating the infrastructure of **software** applications
Author: Owen, Cherry Keahey
Degree: Ph.D.
Year: 2002
Corporate Source/Institution: Texas Tech University (0230)
Chair: Donald Joseph Bagert
Source: VOLUME 63/06-B OF DISSERTATION ABSTRACTS INTERNATIONAL.
PAGE 2902. 206 PAGES
Descriptors: COMPUTER SCIENCE
Descriptor Codes: 0984
ISBN: 0-493-70869-3

In the current environment of heterogeneous distributed computer systems, user applications are more dependent than ever on the infrastructure products. These products include various operating systems, network **software**, **device** **drivers**, and other services needed by the user applications. When user applications request services from the operating system and other system **software**, it is desirable for the requests to be satisfied in a predictable manner.

A user **application** most frequently interacts with the infrastructure through C and C++ function calls. This research studied the class of functions defined by s1 × s2 → s1, where s1 and s2 are null terminated strings, and manipulation is done on the two strings to produce a result that is placed in the original space for s1. Through experimentation, it was found that 9 of 25 C and C++ functions tested allowed writing past the allocated space for s1. Whatever variable or data structure happened to be located in the memory following s1 was **overwritten**. Since the string arguments were passed by reference, these memory **overwrites** were undocumented **updates** to the address space of the calling program. Thus the scope of effect for the function was extended, and the result was unpredictable.

A filter has been written to show how memory **overwriting** can be eliminated for functions in the s1 × s2 → s1 class. The filter truncates the result string before **overwriting** occurs. A truncated string is still an error, but the error is returned through the defined interface for the function, and is predictable. This type of error does not extend the effect, or scope, of the function. Thus the error can be constrained to a defined subset of the **application**, and debugging will be much easier.

The filter is currently implemented for only two functions. However, in future research, the concept could be used to build a megafilter to handle all functions of the s1 × s2 → s1 class. Eventually other classes of functions could be included, and research could be done toward development of a metafilter to handle more than one class of functions.

21/5/4 (Item 1 from file: 56)
DIALOG(R)File 56:Computer and Information Systems Abstracts
(c) 2006 CSA. All rts. reserv.

0000022866 IP ACCESSION NO: 0098685
Low-Cost Aid for Turnkey System Development Pragmatic Designs' DBM-1 Debug
M
Edelson, R H
INTERFACE AGE, v 6, n 3, p 52-54, 1981
PUBLICATION DATE: 1981
DOCUMENT TYPE: Journal Article
RECORD TYPE: Abstract
LANGUAGE: English
FILE SEGMENT: Computer & Information Systems Abstracts

ABSTRACT:

A number of us computer hackers actually use microprocessors to develop "turnkey" or special purpose systems for sale, or as part of a larger piece of equipment. This, besides dedicated hardware configured for the specific task, requires a thoroughly debugged program resident in Prom or ROM. Since

hardware and **software** development are interactive, there should be some method to quickly change the **software** stored in the ROM space as development progresses. What is needed is a memory that can be easily **updated** or changed during operation, but cannot be inadvertently **overwritten** by the development hardware. Pragmatic Designs, Inc. (Sunnyvale, CA) has a **device**, DBM-1 debug memory, that allows just such quick, nonpermanent modifications without the danger of program loss.

DESCRIPTORS: Program development; Development hardware; Turnkey systems; Microprocessors; Storage; **Overwriting**; Dbm-1
SUBJ CATG: C CE3.7, MICROPROCESSORS; MICROCOMPUTERS

22/5/8 (Item 8 from file: 8)
DIALOG(R)File 8:Ei Compendex(R)
(c) 2007 Elsevier English Info. Inc. All rts. reserv.

07247135 E.I. No: EIP95092850200
Title: Fault-tolerant protocol for location directory maintenance in mobile networks
Author: Rangarajan, Sampath; Ratnam, Karunaharan; Dahbura, Anton T.
Corporate Source: Northeastern Univ, Boston, MA, USA
Conference Title: Proceedings of the 25th International Symposium on Fault-Tolerant Computing
Conference Location: Pasadena, CA, USA Conference Date: 19950627-19950630
Sponsor: IEEE
E.I. Conference Number: 43520
Source: Digest of Papers - International Symposium on Fault-Tolerant Computing 1995. IEEE, Piscataway, NJ, USA,95CH35823. p 164-173
Publication Year: 1995
CODEN: DPFTDL ISSN: 0731-3071
Language: English
Document Type: CA; (Conference Article) Treatment: T; (Theoretical)
Journal Announcement: 9511W1
Abstract: In this paper, we present a fault-tolerant protocol for maintaining location directories in mobile networks. The protocol tolerates base station failures and also allows for consistent location information to be maintained about mobile hosts that switch off and arbitrarily reappear in some other part of the network. Further, the protocol tolerates the corruption of a logical time stamp that is part of any protocol where new location information has to be distinguished from old location information when a location directory is **updated**. We formally show that the protocol maintains consistent location information and does not **overwrite** new location information with old location information. The protocol can be hierarchically organized to reduce the message overhead incurred by location directory **updates**. (Author abstract) 15 Refs.
Descriptors: *Fault tolerant computer systems; Mobile radio systems;

Network protocols; Hierarchical systems; Data structures; Computational complexity; Distributed computer systems; **File** organization; Computer networks

Identifiers: Fault tolerant protocol; Location directory maintenance; Message overhead

Classification Codes:

722.4 (Digital Computers & Systems); 716.3 (Radio Systems & Equipment); 723.2 (Data Processing); 721.1 (Computer Theory, Includes Formal Logic, Automata Theory, Switching Theory, Programming Theory)

722 (Computer Hardware); 716 (Radar, Radio & TV Electronic Equipment); 723 (Computer Software); 721 (Computer Circuits & Logic Elements)

72 (COMPUTERS & DATA PROCESSING); 71 (ELECTRONICS & COMMUNICATIONS)

22/5/9 (Item 9 from file: 8)
DIALOG(R)File 8:Ei Compendex(R)
(c) 2007 Elsevier English Info. Inc. All rts. reserv.

06814792 E.I. No: EIP94031227030
Title: Consistency and performance in on-**overwrite** distributed databases
Author: Proszynski, Piotr W.; Eberbach, Eugeniusz
Corporate Source: Coll of Geographic Sciences, Lawrencetown, NS, Can
Conference Title: Proceedings of the 16th Annual Energy-Sources Technology Conference and Exhibition
Conference Location: Houston, TX, USA Conference Date: 19930131-19930204
Sponsor: ASME
E.I. Conference Number: 19938
Source: Computer Applications and Design Abstraction American Society of Mechanical Engineers, Petroleum Division (Publication) PD v 49 1993. Publ by ASME, New York, NY, USA. p 95-101
Publication Year: 1993
CODEN: ASMPEX ISBN: 0-7918-0943-9
Language: English
Document Type: CA; (Conference Article) Treatment: A; (Applications); T ; (Theoretical); X; (Experimental)
Journal Announcement: 9404W4
Abstract: A pragmatic access/ **update** protocol for a distributed multiversion database system is proposed. The protocol guarantees simplicity of recovery process, practically fault-tolerant behavior, and very good performance in highly parallel environments. (Author abstract) 7 Refs.
Descriptors: *Distributed database systems; **File** organization; Computer applications
Identifiers: Multiversion database; **Overwrite** approach; Distributed databases; Recovery process
Classification Codes:
723.3 (Database Systems); 723.5 (Computer Applications)
723 (Computer Software)

72 (COMPUTERS & DATA PROCESSING)

22/5/11 (Item 1 from file: 35)
DIALOG(R)File 35:Dissertation Abs Online
(c) 2006 ProQuest Info&Learning. All rts. reserv.

01651852 ORDER NO: AAD98-37793
SUPPORT FOR MULTI-VIEWED INTERFACES (USER INTERFACE)
Author: SMITH, IAN EMERY
Degree: PH.D.
Year: **1998**
Corporate Source/Institution: GEORGIA INSTITUTE OF TECHNOLOGY (0078)
Director: SCOTT E. HUDSON
Source: VOLUME 59/06-B OF DISSERTATION ABSTRACTS INTERNATIONAL.
PAGE 2862. 149 PAGES
Descriptors: COMPUTER SCIENCE
Descriptor Codes: 0984

The dissertation work addresses the area of the automatic maintenance of relationships. In particular, this work addresses the area of maintaining relationships between user interfaces. We refer to related interfaces as views. This dissertation work gives programmers the ability to declaratively specify the a relationship between one interface--which we call the source view--and another interface--the target view--and have that relationship maintained automatically by the system. For example, suppose that two people wish to work together, but they have different size displays. One person has a traditional desktop workstation and the other has a hand-held computer. A programmer that is building an **application** for these two users can use our experimental system, Ultraman, to declare relationships between to the two interfaces. When the resulting system is in the programmer's relationship(s) between the source and target views are maintained automatically. Crucial to our work, and ignored by previews work in this area, is that these two views need not be exactly the same or even structured similarly. Ultraman allows for the two

views to display semantically related but not necessarily visually identical, information.

Our work can be differentiated from previous work along two primary axes. First, our work focuses on maintaining relationships between trees. Since virtually all modern user interfaces are constructed with a tree structure underlying them, our system allows an **application** developers to specify the relationships between the structures that make up the working interfaces. This is related to, but substantially different from, previous work in the area of constraints in user interfaces, which has focused exclusively on how to maintain the relationships, in the form of equations, between variables. The other way in which this research differs from previous work, primarily from the model-based user interfaces community, is that our transformations of one view into another are performed dynamically, as the program runs. Previous work has been done on giving developers the ability to transform interfaces, but unlike our system, the previous research has focused on performing this transformation only once, at the time the **application** is developed.

A primary technical challenge of this work was that ability to maintain state in a target view that is being maintained automatically based on a source view. Since we transform the source view into the target view each time the source view is changed in the running **application**, how can the target view contain data which is not **overwritten** each time the target view is **updated**? This work presents a novel approach to solving this problem, based on several ideas from the compiler community. Our system maintains an extra copy of the target view temporarily so that the "old version" and the "new version" of a target interface can be compared. This comparison is performed and parts of the old and new versions are grafted together to form the final view shown on the user's screen. This grafting allows portions of the old version that contained interesting state to be preserved and manifested in the final result.

22/5/12 (Item 2 from file: 35)
DIALOG(R)File 35:Dissertation Abs Online
(c) 2006 ProQuest Info&Learning. All rts. reserv.

01315898 ORDER NO: AAD93-30748

SYSTEM SUPPORT FOR **SOFTWARE** FAULT TOLERANCE IN HIGHLY AVAILABLE
DATABASE MANAGEMENT SYSTEMS

Author: SULLIVAN, MARK PAUL
Degree: PH.D.
Year: **1992**
Corporate Source/Institution: UNIVERSITY OF CALIFORNIA, BERKELEY (0028)
Chair: MICHAEL STONEBRAKER

Source: VOLUME 54/06-B OF DISSERTATION ABSTRACTS INTERNATIONAL.
PAGE 3195. 253 PAGES
Descriptors: COMPUTER SCIENCE
Descriptor Codes: 0984

Today, **software** errors are the leading cause of outages in fault tolerant systems. System availability can be improved despite **software** errors by fast error detection and recovery techniques that minimize total downtime following an outage. This dissertation analyzes **software** errors in three commercial systems and describes the implementation and evaluation of several techniques for error detection and fast recovery in a database management system (DBMS).

The **software** error study examines errors reported by customers in three IBM systems programs: the MVS operating system, the IMS DBMS, and the DB 2 DBMS. The study classifies errors by the type of coding mistake and the circumstances in the customer's environment that caused the error to arise. It observes a higher availability impact from addressing errors, such as uninitialized pointers, than **software** errors as a whole. It also details the frequencies and types of addressing errors and characterizes the damage they do.

The error detection work evaluates the use of hardware write protection both to detect addressing-related errors quickly and to limit the damage that can occur after a **software** error. System calls added to the operating system allow the DBMS to guard (write-protect) some of its internal data structures. Guarding DBMS data provides quick detection of corrupted pointers and similar **software** errors. Data structures can

be guarded as long as correct **software** is given a means to temporarily unprotect the data structures before **updates**. The dissertation analyzes the effects of three different **update** models on performance, **software** complexity, and error protection.

To improve DBMS recovery time, previous work on the POSTGRES DBMS has suggested using a storage system based on no-**overwrite** techniques instead of write-ahead log processing. The dissertation describes modifications to the storage system that improve its performance in environments with high **update** rates. Analysis shows that, with these modifications and some non-volatile RAM, the I/O requirements of POSTGRES running a TP1 benchmark will be the same as those of a conventional system, despite the POSTGRES force-at-commit buffer management policy. The dissertation also presents an extension to POSTGRES to support the fast recovery of communication links between the DBMS and its clients.

Finally, the dissertation adds to the fast recovery capabilities of POSTGRES with two techniques for maintaining B-tree index consistency without log processing. One technique is similar to shadow paging, but improves performance by integrating shadow meta-data with index meta-data. The other technique uses a two-phase page reorganization scheme to reduce the space overhead caused by shadow paging. Measurements of a prototype implementation and estimates of the effect of the algorithms on large trees show that they will have limited impact on data manager performance.

22/5/13 (Item 3 from file: 35)
DIALOG(R)File 35:Dissertation Abs Online
(c) 2006 ProQuest Info&Learning. All rts. reserv.

01175247 ORDER NO: AAD91-28371
A DATA-FLOW MODEL OF REAL-TIME SYSTEMS (SCHEDULING)
Author: SUTANTHAVIBUL, SUPOJ
Degree: PH.D.
Year: **1991**
Corporate Source/Institution: THE UNIVERSITY OF TEXAS AT AUSTIN (0227)
Supervisor: ALOYSIUS K. MOK

Source: VOLUME 52/04-B OF DISSERTATION ABSTRACTS INTERNATIONAL.
PAGE 2152. 116 PAGES
Descriptors: COMPUTER SCIENCE
Descriptor Codes: 0984

We present an AND-OR data-flow model for a class of real-time systems in which input data arrive at fixed rates. There is no explicit timing constraint except that the loss of data, whether coming from external sources or generated within the system, is not allowed. Data loss may occur even if the processor is underutilized: whenever a node in the data-flow graph generates data twice from two successive computations, and the latter data **overwrite** the former before they are used by other nodes, then data loss has occurred.

For the basic AND data-flow model, we demonstrate that the nonpreemptive uniprocessor-scheduling problem is NP-hard. However if preemption is allowed, there exist optimal schedulers which are efficient enough to be used on-line. To determine whether a preemptive schedule exists, one just computes the processor utilization factor of the system. When the OR nodes are added, we show that the utilization-factor determination problem is NP-hard. But the scheduling problem remains essentially the same.

We also investigate the issues associated with the distributed systems. We demonstrate that the problem of mapping the data-flow graph onto a simple configuration of two connected processors to minimize the communication **load** is NP-complete, assuming the processor demands of the nodes are identical. The nonpreemptive schedulings of messages, in general, are NP-hard.

In relation to the preemptive message scheduling, we study the case in which the messages make at most two hops in unidirection ring networks. We discover a technique for collapsing messages into a single message called the deputy message. We present an optimal scheduling algorithm for a case in which the periodicities of the double-hop messages are multiples of the period of their deputy message.

One of our goals is to develop the AND-OR data-flow model as a methodology for designing real-time systems. Toward this goal, we apply our model to an actual real-time message processing system. We discover some

shortcomings of the model but we are able to address them successfully. The **application** of the model has proved that our model has practical value.

22/5/18 (Item 5 from file: 2)
DIALOG(R)File 2:INSPEC
(c) 2007 Institution of Electrical Engineers. All rts. reserv.

08144269 INSPEC Abstract Number: C2002-02-6150C-024
Title: Unroll-based register coalescing
Author(s): Suhyun Kim; Soo-Mook Moon; Jinpyo Park; Ebciolu, K.
Author Affiliation: Sch. of Electr. English, Seoul Nat. University, South Korea
Conference Title: Conference Proceedings of the 2000 International Conference on Supercomputing p.296-305
Publisher: ACM, New York, NY, USA
Publication Date: 2000 Country of Publication: USA xi+347 pp.
ISBN: 1 58113 270 0 Material Identity Number: XX-2000-01089
U.S. Copyright Clearance Center Code: 1-58113-270-0/00/$5.00
Conference Title: Proceedings of ICS00: International Conference on Supercomputing
Conference Sponsor: ACM
Conference Date: 8-11 May 2000 Conference Location: Sante Fe, NM, USA
Language: English Document Type: Conference Paper (PA)
Treatment: Applications (A); Practical (P)
Abstract: Aggressive instruction scheduling leaves behind many renaming copy instructions that cannot be coalesced due to interferences. These copies take resources, and more seriously, they may cause a stall if they are generated for renaming of multi-latency instructions. This paper proposes a code transformation technique based on loop unrolling which makes those copies coalescible. Two unique features of the technique are its method of determining the precise unroll amount based on an idea of extended live range, and its insertion of special bookkeeping copies at loop exits. In fact, the technique provides a more general and simpler solution for the cross-iteration register **overwrite** problem in **software** pipelining which works for loops with control flows as well

as for straight-line loops. In addition, it is applicable to other optimizations including path length reduction and redundant subscripted reference elimination. Our empirical study Performed on a 16-ALU VLIW testbed with a two-cycle **load** latency shows that 86% of the otherwise uncoalescible copies in innermost loops become coalescible when unrolled 2.2 times on average. In-addition, it is demonstrated that the unroll amount obtained is precise and the most efficient. The unrolled version of the VLIW code includes fewer no-op VLIWs caused by stalls, improving the performance by a geometric mean of 18%. (10 Refs)

Subfile: C

Descriptors: compiler generators; instruction sets; pipeline processing; processor scheduling; **software** performance evaluation

Identifiers: unroll-based register coalescing; aggressive instruction scheduling; renaming copy instructions; multi-latency instructions; code transformation technique; loop unrolling; extended live range; cross iteration register **overwrite** problem; **software** pipelining; 16-ALU VLIW testbed; two-cycle **load** latency; VLIW code; performance evaluation

Class Codes: C6150C (Compilers, interpreters and other processors); C6150N (Distributed systems software); C5440 (Multiprocessing systems)

Copyright 2002, IEE

22/5/19 (Item 6 from file: 2)
DIALOG(R)File 2:INSPEC
(c) 2007 Institution of Electrical Engineers. All rts. reserv.

07966685 INSPEC Abstract Number: C2001-08-6150J-024

Title: Embedded **software** using UPM (Universal Process Model) operating systems

Author(s): Stumpf, M.

Author Affiliation: QNX Software Syst. GmbH, Hannover, Germany

Journal: Elektronik Praxis no.11 p.86-90

Publisher: Vogel-Verlag,

Publication Date: 6 June 2001 Country of Publication: Germany

CODEN: EKPXAM ISSN: 0341-5589

SICI: 0341-5589(20010606)11L.86:ESUU;1-I

Material Identity Number: E248-2001-012

Language: German Document Type: Journal Paper (JP)

Treatment: Applications (A); Practical (P)

Abstract: Describes **software** reliability problems caused by "flat" or monolithic architecture, where each **software** module is capable of **over-writing** the main memory. Notes that this architecture can provide protection only for individual applications, but not for the monolithic kernel. Illustrates how the "Universal Process Model" (UPM) employs a central micro-kernel only for essential common services, such as scheduling, communications and routing of hardware interrupts, whereas other services are provided by individual self-contained processes. This is stated to allow starting, stopping, modification or **upgrading** of the individual protected **software** applications modules, without need for re-starting, re-programming or re-testing of the central program kernel. It is stated that this allows entirely independent programming of the system modules, so that development can be split into independent projects. It is claimed that UPM permits automatic rebuilding after **software** faults, as well as rapid exchange of hardware or **software** modules. The use of separate "**software** watchdogs" is recommended, to allow " **Software** Hot Swapping". Also reports that UPM facilitates partitioning of applications between several CPUs. Refers to the QNX UPM operating system, which can stop and re-start a process on a 133 MHz Pentium microprocessor in under 2 microseconds. Announces a QNX developer network. (0 Refs)

Subfile: C

Descriptors: interrupts; operating system kernels; **software** reliability

Identifiers: UPM; Universal Process Model; operating systems; **software** reliability problems; **software** module; monolithic kernel; central micro-kernel; scheduling; hardware interrupts; self-contained processes; protected **software** applications modules; automatic rebuilding; **software** watchdogs; **Software** Hot Swapping ; QNX; embedded **software**

Class Codes: C6150J (Operating systems); C6110B (Software engineering techniques)

Copyright 2001, IEE

22/5/20 (Item 7 from file: 2)
DIALOG(R)File 2:INSPEC
(c) 2007 Institution of Electrical Engineers. All rts. reserv.

07912227 INSPEC Abstract Number: C2001-06-6160J-007
Title: A comparative study of log-only and in-place **update** based temporal object database systems
Author(s): Norvag, K.
Author Affiliation: Dept. of Comput. & Inf. Sci., Norwegian Inst. of Technol., Trondheim, Norway
Conference Title: Proceedings of the Ninth International Conference on Information and Knowledge Management. CIKM 2000 p.496-503
Editor(s): Agah, A.; Callan, J.; Rundensteiner, E.
Publisher: ACM, New York, NY, USA
Publication Date: 2000 Country of Publication: USA xvi+532 pp.
ISBN: 1 58113 320 0 Material Identity Number: XX-2000-03146
U.S. Copyright Clearance Center Code: 1 58113 320 0/2000/0011...$5.00
Conference Title: Proceedings of Ninth International Conference on Information and Knowledge Management (CIKM)
Conference Sponsor: ACM
Conference Date: 6-11 Nov. 2000 Conference Location: McLean, VA, USA
Language: English Document Type: Conference Paper (PA)
Treatment: Practical (P)
Abstract: In most current database systems, data is **updated** in-place. In order to support recovery and increase performance, write-ahead logging is used. This logging defers the in-place **updates**, however sooner or later, the **updates** have to be applied to the database. This often results in non-sequential writing of lots of pages, creating a write bottleneck. To avoid this, another approach is to eliminate the database completely, and use a log-only approach. The log is written contiguously to the disk, in a no-**overwrite** way, in large blocks. The log-only approach is particularly interesting for transaction-time object database systems (TODBs). While previous approaches to TODBs have been page based, i.e., when an object has been modified, the whole page the object resides on has to be written back, our approach is object based. One of the objections against operating at object granularity

is that the read cost will be prohibitively high. We show that this is not necessarily true. We use analytical cost models to compare the performance of log-only and in-place **update** TODBs, and the analysis shows that with the workload we expect to be typical for future TODBs, the log-only approach is highly competitive with the traditional in-place **update** approach. (12 Refs)
Subfile: C
Descriptors: object-oriented databases; query processing; **software** performance evaluation; temporal databases
Identifiers: temporal object database systems; in-place data **update** ; write-ahead logging; nonsequential writing; write bottleneck; log-only approach; transaction-time object database; object granularity; analytical cost models
Class Codes: C6160J (Object-oriented databases); C6160Z (Other DBMS)
Copyright 2001, IEE

22/5/21 (Item 8 from file: 2)
DIALOG(R)File 2:INSPEC
(c) 2007 Institution of Electrical Engineers. All rts. reserv.

07691683 INSPEC Abstract Number: C2000-10-6160Z-008
Title: A performance evaluation of log-only temporal object database systems
Author(s): Norvag, K.
Author Affiliation: Dept. of Comput. & Inf. Sci., Norwegian University of Sci. & Technol., Trondheim, Norway
Conference Title: Proceedings. 12th International Conference on Scientific and Statistica Database Management p.256-8
Editor(s): Gunther, O.; Lenz, H.-J.
Publisher: IEEE Comput. Soc, Los Alamitos, CA, USA
Publication Date: 2000 Country of Publication: USA x+261 pp.
ISBN: 0 7695 0686 0 Material Identity Number: XX-2000-01866
U.S. Copyright Clearance Center Code: 0 7695 0686 0/2000/$20.00
Conference Title: Proceedings. 12th International Conference on Scientific and Statistical Database Management
Conference Sponsor: Deutsche Forschungsgemeinschaft; Freie University Berlin; Humbolt-University zu Berlin; MicroStrategy Deutschland; SAS Inst. Deutschland;

Scopeland; Statistisches Landesamt Berlin
Conference Date: 26-28 July 2000 Conference Location: Berlin, Germany
Language: English Document Type: Conference Paper (PA)
Treatment: Practical (P)
Abstract: An alternative to in-place **updating** of data is to eliminate the database completely, and use a log-only approach. The log is written contiguously to the disk, in a no-**overwrite** way, in large blocks. The log-only approach is particularly interesting for transaction time object database systems (TODB), because keeping previous versions of objects is a feature that comes for free. One of the objections against log-only databases has been that the read cost will be prohibitively high because of loss of clustering. However in our comparison of the performance of log-only and in-place **update** TODBs, the analysis shows that with the workload we expect to be typical for future TODBs, the log-only approach is highly competitive with the traditional in-place **update** approach. (7 Refs)
Subfile: C
Descriptors: object-oriented databases; **software** performance evaluation; temporal databases
Identifiers: log-only; temporal object database systems; transaction time object database systems; performance evaluation; workload
Class Codes: C6160Z (Other DBMS); C6160J (Object-oriented databases)
Copyright 2000, IEE

22/5/23 (Item 10 from file: 2)
DIALOG(R)File 2:INSPEC
(c) 2007 Institution of Electrical Engineers. All rts. reserv.

06859569 INSPEC Abstract Number: C9804-7140-242
Title: Cognition-based development and evaluation of ergonomic user interfaces for medical image processing and archiving systems
Author(s): Demiris, A.M.; Meinzer, H.P.
Author Affiliation: German Cancer Res. Center, Heidelberg, Germany
Journal: Medical Informatics Conference Title: Med. Inform. (UK) volume22, no.4 p.349-58
Publisher: Taylor & Francis,
Publication Date: Oct.-Dec. 1997 Country of Publication: UK
CODEN: MINFDZ ISSN: 0307-7640
SICI: 0307-7640(199710/12)22:4L.349:CBDE;1-4
Material Identity Number: I877-98001
U.S. Copyright Clearance Center Code: 0307-7640/97/$12.00
Conference Title: EuroPACS. 14th International Meeting
Conference Date: 3-5 Oct. 1996 Conference Location: Heraklion, Greece
Language: English Document Type: Conference Paper (PA); Journal Paper (JP)
Treatment: Practical (P)
Abstract: Whether or not a computerized system enhances the conditions of work in the **application** domain, very much demands on the user interface. Graphical user interfaces seem to attract the interest of the users but mostly ignore some basic rules of visual information processing, thus leading to systems which are difficult to use, lowering productivity and increasing working stress (cognitive and work **load**). In this paper, we present some fundamental ergonomic considerations and their **application** to the medical image processing and archiving domain. We introduce extensions to an existing concept needed to control and guide the development of GUIs with respect to domain-specific ergonomics. The suggested concept, called model-view-controller constraints (MVCC), can be used to programmatically implement ergonomic constraints, and thus has some

advantages **over** **written** style guides. We conclude with the presentation of existing norms and methods to evaluate user interfaces. (21 Refs)
Subfile: C
Descriptors: ergonomics; graphical user interfaces; medical image processing; PACS; user centred design; visual databases
Identifiers: cognition-based development; ergonomic user interface evaluation; medical image processing systems; medical image archiving systems; work conditions; graphical user interfaces; visual information processing; usability; productivity; working stress; cognitive **load**; work **load**; domain-specific ergonomics; model-view-controller constraints; ergonomic constraints; style guides; **software** ergonomics ; user-centered design
Class Codes: C7140 (Medical administration); C5260B (Computer vision and image processing techniques); C7330 (Biology and medical computing); C6160S (Spatial and pictorial databases); C6180G (Graphical user interfaces); C6110 (Systems analysis and programming)
Copyright 1998, IEE

22/5/40 (Item 4 from file: 6)
DIALOG(R)File 6:NTIS
(c) 2007 NTIS, Intl Cpyrght All Rights Res. All rts. reserv.

0749695 NTIS Accession Number: PB-290 836/6/XAB
NEWECL: Economic Computer Language Data Handlers
(Final rept)
Renner, R. L.
Mathematics and Computation Lab. (EDM), Washington, DC. Applied Economics Div.
Report No.: GSA/FPA/MCL-TM-250-REV-2
Dec 78 60p
Languages: English
Journal Announcement: GRAI7911
Supersedes PB-262 128.
Order this product from NTIS by: phone at 1-800-553-NTIS (U.S. customers); (703)605-6000 (other countries); fax at (703)321-8547; and email at orders@ntis.fedworld.gov. NTIS is located at 5285 Port Royal Road,

Springfield, VA, 22161, USA.
NTIS Prices: PC A04/MF A01
Since their introduction in 1972 and revision in 1975 by the Mathematics and Computation Laboratory (MCL), the Economic Computer Language Data Handlers have become a major and integral part in almost all computer implemented economic applications projects on the Univac 1108 computer system at the Federal Preparedness Agency (FPA). The unique capabilities offered by the NEWECL package have been recognized by other groups in the Univac computer community with the result that NEWECL has been transferred to or duplicated on a number of other Univac 1100 series **installations** both inside and outside the Federal Government. The NEWECL system is distributed through the National Technical Information Service. This revision of the NEWECL manual was necessitated by the replacement of the Fieldata version of the BASIC language compiler (RBASIC) on the system libraries with the ASCII version (ABASIC). This change required an **update** to the NEWECL BASIC package. The previous package was compatible with RBASIC programs only, while the **updated** version is now compatible with both RBASIC and ABASIC programs. The packet utilization procedure for ABASIC programs to access NEWECL is different from the procedure for RBASIC programs, which has not changed. The user should refer to the appropriate section of this manual for the new procedure. In addition, several sections of the manual have been rewritten for clarification and the descriptions of two provisions of NEWECL--the **overwrite** protection feature and automatic **file** assignment with read/write keys--have been expanded.
Descriptors: *Programming manuals; Economic analysis; Programming languages; Data processing; Fortran; Basic programming language
Identifiers: NEWECL programming language; Fortran 5 programming language; RBASIC programming language; ABASIC programming language; Univac-1100
computers; NTISMCLAB
Section Headings: 62B (Computers, Control, and Information Theory--Computer Software); 96GE (Business and Economics--General)

22/5/42 (Item 2 from file: 144)
DIALOG(R)File 144:Pascal
(c) 2007 INIST/CNRS. All rts. reserv.

15126171 PASCAL No.: 01-0288651
Simple confluently persistent catenable lists
KAPLAN Haim; OKASAKI Chris; TARJAN Robert E
Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel; Department of Computer Science, Columbia University, New York, NY 10027, United States; Department of Computer Science, Princeton University, Princeton, United States; NJ 08544 and InterTrust Technologies Corporation, Sunny vale, CA 94086, United States
Journal: SIAM journal on computing, **2001**, 30 (3) 965-977
ISSN: 0097-5397 Availability: INIST-16063; 354000098302340100
No. of Refs.: 18 ref.
Document Type: P (Serial) ; A (Analytic)
Country of Publication: United States
Language: English
We consider the problem of maintaining persistent lists subject to concatenation and to insertions and deletions at both ends. **Updates** to a persistent data structure are nondestructive each operation produces a new list incorporating the change, while keeping intact the list or lists to which it applies. Although general techniques exist for making data structures persistent, these techniques fail for structures that are subject to operations, such as catenation, that combine two or more versions. In this paper we develop a simple implementation of persistent double-ended queues (deques) with catenation that supports all deque operations in constant amortized time. Our implementation is functional if we allow memoization.

English Descriptors: Functional programming; Data structure; Stack; Tail; Storage; Confluence; Implementation; List; Concatenation; Insertion; Deletion; End; Queue; Support; Functional; Stack ended queue; Double ended queue; **Overwriting**

French Descriptors: Programmation fonctionnelle; Structure donnee; Pile

memoire; Queue; Stockage; Confluence; Implementation; Liste; Concatenation; Insertion; Deletion; Extremite; **File** attente; Support; Fonctionnelle; Steque; Deque; Surecriture

Classification Codes: 001D02B07B; 001D02A05

Copyright (c) 2001 INIST-CNRS. All rights reserved.

IEEE Xplore RELEASE 2.1

Home | Login | Logout | Access Information | Alerts | Sitemap | Help

Welcome United States Patent and Trademark Office

BROWSE SEARCH IEEE XPLORE GUIDE SUPPORT

e-mail printer friendly

AbstractPlus

View Search Results | Next Article

Access this document

Full Text: PDF (724 KB)

Download this citation

Choose Citation & Abstract

Download ASCII Text

» Learn More

Rights and Permissions

» Learn More

The Galaxy distributed operating system

Sinha, P.K. Maekawa, M. Shimizu, K. Jia, X. Ashihara, H. Utsunomiya, N. Park, K.S. Nakano, H.
Tokyo Univ., Japan;

This paper appears in: Computer
Publication Date: Aug. 1991
Volume: 24 , Issue: 8
On page(s): 34 - 41
ISSN: 0018-9162
CODEN: CPTRB4
INSPEC Accession Number:4002330
Digital Object Identifier: 10.1109/2.84875
Posted online: 2002-08-06 17:53:44.0

Abstract
The Galaxy research project, which is attempting to design, implement, and use a distributed computing environment, based on the idea of making gradual improvements by learning from existing systems and trying to overcome their limitations, is described. The design goals and novel aspects of the project are outlined, and the Galaxy system architecture is described. Galaxy's object **naming** and locating mechanisms, interprocess communication, and computation model are discussed

Index Terms
Inspec
Controlled Indexing
distributed processing network operating systems

Non-controlled Indexing
Galaxy distributed operating system Galaxy system architecture computation model distributed computing environment interprocess communication locating mechanisms object naming
Author Keywords
Not Available

References

No references available on IEEE Xplore.

Citing Documents

No citing documents available on IEEE Xplore.

View Search Results | Next Article

Help Contact Us Privacy & Security IEEE.org

Indexed by Inspec

© Copyright 2006 IEEE – All Rights Reserved

The Galaxy Distributed Operating System

Pradeep K. Sinha, Mamoru Maekawa, Kentaro Shimizu, Xiaohua Jia, Hyo Ashihara, Naoki Utsunomiya, Kyu S. Park, and Hirohiko Nakano
University of Tokyo

Analyzing why existing distributed systems are limited, the Galaxy research project borrows many concepts and proposed facilities and integrates them with its own novel design features.

Judging from the enormous amount of distributed-system research carried out over the past decade, information processing experts have come to recognize the advantages these systems possess. These research activities have led to the availability of more than 50 network and distributed systems. However, most of these systems can only partially succeed in attaining the major goals of a distributed system, which include transparency, higher performance, higher reliability and availability, and higher scalability. Of course, attaining all these goals in the first attempt is impossible.

Nonetheless, gradual improvements are possible by learning from existing systems and trying to overcome their limitations. The University of Tokyo's Galaxy research project attempts to design, implement, and use a distributed computing environment based on this idea.

Design goals and novel aspects

In designing Galaxy, our primary goal was to build a distributed operating system suitable for use in both local and widearea network workstations. To achieve this goal, we eliminated the use of broadcast protocols in any of our mechanisms, avoided the use of global-locking or time-stamp-ordering mechanisms while maintaining the consistency of replicated information, and decentralized the management of all globally useful information.

Our second major goal was to ensure high performance. Design decisions and novel system features that helped us achieve this goal include

- developing the Galaxy kernel from scratch, rather than modifying an existing Unix kernel;
- using multiple-level interprocess communication mechanisms to meet the various communication needs of different applications;

0018-9162/91/0800-0034$01.00 © 1991 IEEE

• employing the concept of variable-weight processes to allow flexibility and efficiency in data sharing and process scheduling; and
• inventing an ID-table-based direct object locating mechanism.

Our next major goal is higher reliability and availability. Our direct object locating mechanism, user-definable reliability parameters for name resolution operation, use of data caches and name caches, and file replication mechanism help in improving our system's overall reliability and availability.

Another of our design goals is to make Galaxy's application interface compatible with that of Unix. To this end, we first designed the best interface from the viewpoint of distributed operating systems and then properly modified it to make it a superset of the present Unix interface. In addition, to achieve this goal, and as long as other goals did not suggest using any special techniques, we modeled our system as closely as possible to Unix.

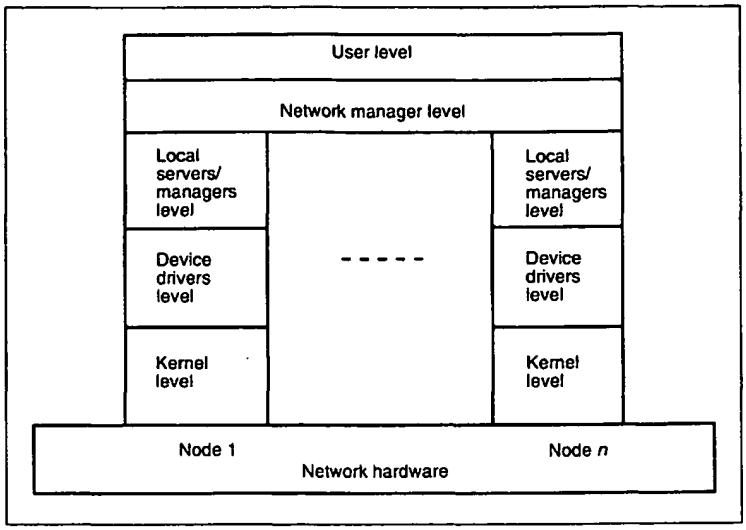

Figure 1. Galaxy's layered system structure.

Galaxy system architecture

The Galaxy design belongs to the server-pool class of systems, with an underlying computational model based on objects. We had two main reasons for adopting the server-pool approach. First, Galaxy's design allows for the existence of diskless workstations in the network. This design issue precludes the use of an integrated system approach in which each node of the network is an autonomous, stand-alone system. Our second reason is our design policy of placing a particular module of the system only on those nodes of a network where there is some possibility of using it. Blindly maintaining all system utilities at every node appears to waste various system resources.

Galaxy objects. In Galaxy, the entities that need to be identified at the operating system level (such as processes, files, devices, and nodes) are viewed as objects. Every object in Galaxy belongs to a particular type. A type describes a set of objects with the same characteristics. A type also describes the structure of data carried by objects as well as the operations (*methods* in the object-oriented terminology) applied to these objects. Users of a type see only the interface of the type, that is, a list of methods together with their signatures (the type of input parameters and the type of the result).

In Galaxy, each type of object is managed by a special module dedicated to the type. We call this module the *object manager*. A particular type of object manager resides on all the nodes where objects of that type exist. When an operation invocation message is issued to an object, the corresponding object manager is invoked.

This object management policy has several attractive features:

(1) It ensures that access to managed resources (objects) is efficient, because the manager and the managed object reside on the same node.

(2) It ensures that the presence of an object manager at a particular node is not wasteful.

(3) It eases the problem of coping with hardware failures since, for example, if a node crashes, the managers for all the objects of that node will also be unavailable.

(4) It is easily adaptable to different hardware configurations.

System structure. One of the most important issues in designing the structure of a distributed operating system is the size of the kernel that is replicated at each node of the network. In a large-kernel approach, taken by such systems as Locus[1] and Sprite,[2] services are provided more efficiently than in cases where they are offered outside the kernel. However, this approach reduces the overall flexibility and configurability of the resulting operating system.

In a small-kernel approach, taken by V,[3] most of the services are executed by separate servers. The servers are usually implemented as processes and can be programmed separately. A small kernel provides the process scheduling, interprocess communication, and some other primitive operations. This approach accomplishes the maximum flexibility, but it degrades efficiency.

In Galaxy, we use the small-kernel approach. For the sake of efficiency, we structure our system in multiple levels with various communication facilities between the processes of these levels. As Figure 1 shows, Galaxy features five levels:

(1) the kernel level,
(2) the device drivers level,
(3) the local servers/managers level,
(4) the network manager level, and
(5) the user level.

The kernel level, which is replicated on all the nodes, contains the most primitive functions of the system. These functions are implemented as procedures in

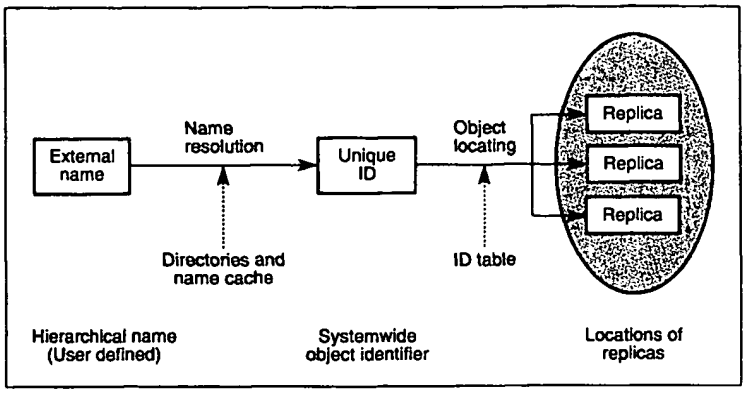

Figure 2. Galaxy's three-level naming scheme.

the kernel and can be classified into the following types:

(1) interrupt-handling routines, which set commands to hardware and catch interrupt signals from hardware,

(2) routines for handling the most primitive interprocess communications,

(3) process-handling routines, which manage system structure for processes, save and restore process context, and perform process scheduling, and

(4) memory management routines, which manage physical memory and perform read/write operations on the physical memory.

The device drivers level consists of the device driver routines, including clock driver, disk driver, tty driver, and network driver. The availability of a particular routine of this level at a particular node depends on the availability of the corresponding device on that node. The device drivers communicate with the kernel frequently by sending kernel input/output commands and responding to the kernel's interrupts. Thus, they share the address space with the kernel, so that they can access kernel data structure and respond to the kernel faster. The routines in the kernel level and the device drivers level run in privileged mode.

The routines at the local servers/managers level perform local object management. For example, the file manager routine is responsible for handling the local files, the process manager routine is responsible for handling the local processes, etc. The ID manager routine of this level provides the location of a remote object for the realization of location-independent object accessing in our system.

The routines at the network manager level perform networkwide object management. Networkwide virtual memory, object migration, object replication, and the like are realized at this level. Most modules at the local servers/managers level and the network manager level are implemented as processes that have separate address spaces.

The user level provides services such as object naming, user management, and implementation of object-based environment. The modules at this level are implemented as distributed servers; they can be located at any node, and their programs are location independent. Other modules at this level are implemented as local servers, but a user can invoke both categories of modules by using the same interprocess communication facility.

Object naming and locating

In designing Galaxy's object-naming and object-locating mechanisms, we were particularly concerned about transparency, scalability, reliability, and efficiency issues. To satisfy the transparency requirement, we used the single global name space model used by such recent distributed systems as Locus,[1] V,[4] and Sprite,[5] because it supports location transparency of both request-receiving objects and request-issuing objects.

Direct mapping of a user-defined object name to the object's location every time the object is accessed will require the name resolution operation to be carried out for every access to the object, which will degrade efficiency. Moreover, user-defined names are not unique for a particular object and are variable in length, not only for different objects but even for the different names of the same object. Hence, they cannot be easily manipulated, stored, and used by the machines for identification. Therefore, in addition to user-defined hierarchical names, which are useful for the users, Galaxy employs system-defined flat names that can be used efficiently by the system.

In Galaxy, user-defined names are called *external names*, and system-defined names are called *unique identifiers* (ID for short). As Figure 2 shows, an external name is first mapped to its corresponding ID, which in turn is mapped to the physical locations (node numbers) of the replicas of the concerned object. The Galaxy process of mapping an external name to its corresponding ID is called *name resolution*, and the process of mapping an ID to the physical locations of the replicas of the concerned object is known as *object locating*.

The object-locating mechanism. The general requirement for object locating is a mapping table (called ID table in Galaxy), each entry of which consists of the locating information of an object. In particular, a Galaxy ID-table entry (or IDTE) contains information about the type of the object, an access control list for the object, locations of the object's replicas (*replica list*), and locations where the copies of this IDTE exist (*copy list*).

The replica list helps in returning all the locations of the desired object as a result of the object-locating operation. Using the copy list, all IDTEs of the same object are linked together so that any modification can be consistently made to all copies.

Thus, given an object's ID, its physical location can be known simply by searching the given ID in the ID table and extracting the physical locations of its replicas from the ID table. However, the most difficult facet of the locating mechanism is the decision policy for maintaining this mapping table.

Several methods have been proposed and used in existing distributed systems:

- the method of broadcasting,
- the method of hint cache and broadcasting,
- the method of chaining,
- the centralized server method in which the entire ID table is kept on a single node, and
- the full replication method in which the entire ID table is replicated on all the nodes

However, all these mechanisms suffer from one or more of the common limitations of poor efficiency, poor reliability, and poor scalability.

Galaxy uses a totally different approach to overcome the limitations of the existing mechanisms. On a particular node, Galaxy keeps only the locating information for objects that have some possibility of being accessed from the concerned node. To achieve this, we determined that in Galaxy only the following two categories of IDs are necessary in the ID table of a particular node:

(1) IDs contained in a directory or in a name cache of that node. The presence of these IDTEs in the local ID table of a node is necessary for the direct locating of the directory file objects during the name resolution process.

(2) IDs being used by the processes running on the node. These IDTEs are necessary in a node's ID table for the direct locating of the objects being used by that node's processes.

Based on this idea of direct-object locating and our object management policy discussed in the section entitled "Galaxy objects," a process accesses a remote object as illustrated in Figure 3. This approach enjoys the advantages of very high efficiency, reliability, and scalability of the object-locating operation because any object can be located directly by using the locating information stored in the local node without the need to replicate the entire ID table on all the nodes.

We determined the following primitives to be sufficient for IDTE management: ReadIDTE(*ID,field*), InsertItem(*ID, field, item*), and DeleteItem(*ID, field, item*). We observed that each IDTE is a collection of fields and that each field can be updated incrementally as well as independently. We also found that the two update operations *InsertItem* and *DeleteItem* are commutative operations; that is, the final result produced by a sequence of these update operations does not depend on the order in which the operations are executed.

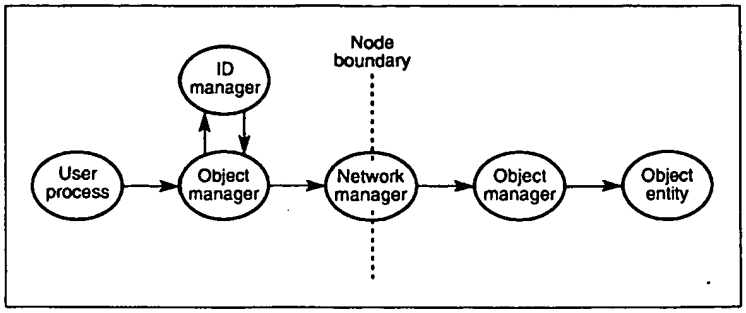

Figure 3. Remote-object-accessing mechanism.

The execution order of the two update operations should be serialized only when two update operations are performed on the same item in the same field of the same IDTE. These properties made it possible to design an update mechanism on replicated IDTEs with high concurrency and efficiency.

In this mechanism, read and update operations are done directly to the local replica of the ID table. The system executes an update operation issued by a local user on the local replica and returns control to the user immediately. The update operation is propagated later to other nodes holding a replica, with no need of synchronization.

To allow updates to an IDTE even during the creation of a new replica of the IDTE, we define the node that makes a new replica of an IDTE at another node to be the parent of the newly created replica. The node in which the parent resides must keep informing the other replicas of the IDTE until they all come to know about the newly created replica. The access consistency is checked when the location information in an IDTE is used to locate the corresponding object.

Compared with other concurrency and consistency control mechanisms that guarantee strict consistency among replicas of data, our approach has the following characteristics:

(1) neither global locking nor timestamp ordering is required.

(2) updating operations can be issued and executed without being synchronized, and

(3) accesses to directories are allowed even during insertion of a new object replica or an IDTE replica as well as during the deletion of an object replica or an IDTE replica.

Jia et al.[6] discuss the correctness and applicability of the algorithm.

The name resolution mechanism. In conventional operating systems, directories are used to map an object's name to its physical location. In these systems, a directory entry consists of a component name and the corresponding object descriptor pointer, such as *i*-nodes in Unix[7] and *v*-nodes in the Sun NFS.[8] Galaxy differs from those systems in that a Galaxy directory entry is a (*component name, ID*) pair that maps a component name of an object to its system-wide unique ID. In addition, in Galaxy, *name caches* are used at each node for caching recently used directory entries. Like the directories, Galaxy's name caches are hierarchical in structure. Using the name cache and the directory objects, the name resolution operation is performed by the basic method of remote pathname expansion.[9]

In the remote pathname expansion mechanism, the reliability of resolving a pathname from a particular node is greatly influenced by the locations of the directories or their replicas that correspond to the given pathname components. On the basis of the various possible locations of the replicas of the concerned directories, we define the following factors that influence the reliability of the name resolution operation:

(1) *Subpath:* For a (node, pathname) pair, the subpath reliability of the name

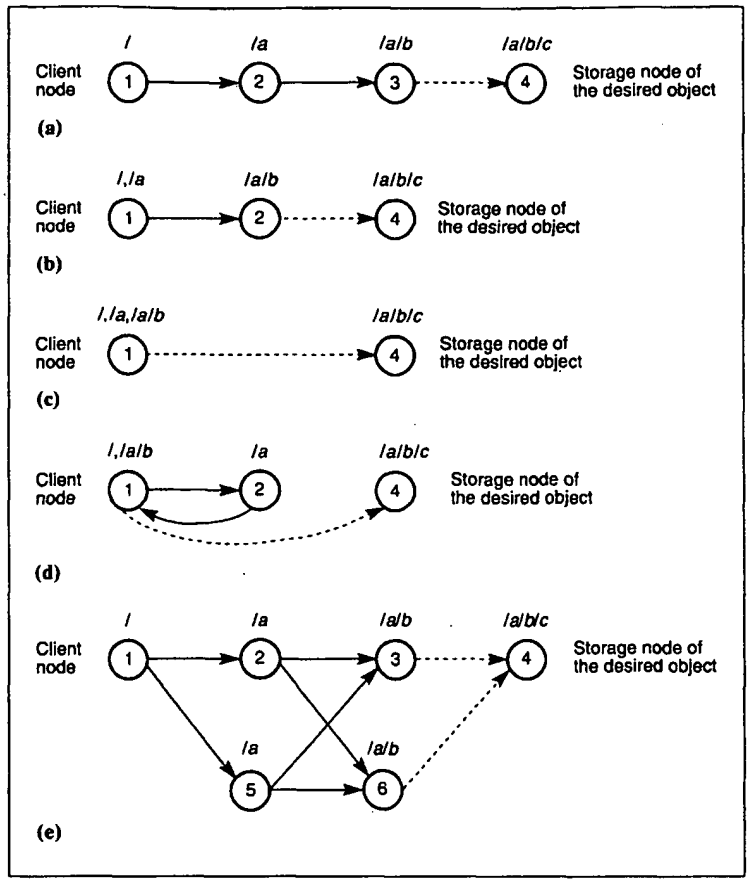

Figure 4. Typical examples of name resolution reliability factors while locating an object with pathname /a/b/c: (a) values of reliability factors are subpath =/, *m*-stage = 2, *k*-path = 1; (b) values of reliability factors are subpath = /a, *m*-stage = 1, *k*-path = 1; (c) values of reliability factors are subpath = /a/b, *m*-stage = 0, *k*-path = ∞; (d) values of reliability factors are subpath = /, *m*-stage = 2, *k*-path = 1; (e) values of reliability factors are subpath = /, *m*-stage = 2, *k*-path = 2.

resolution operation is the presence of the necessary directories on the client node (specified node) for tracing the components of the pathname up to the subpath right at the client node without communicating with any other node.

(2) *m-stage:* A name resolution operation is said to be *m*-stage unreliable when *m* hops are required during pathname expansion for resolving the given pathname. In our method of remote pathname expansion, a *hop* is defined as the passing of the remaining pathname components from one node to another node, which has the next component's directory when the remaining components of the pathname cannot be further resolved on the present node.

(3) *k-path:* A name resolution operation is said to be *k*-path reliable when there are at least *k* possible paths between any two contiguous directories corresponding to the components of the given pathname starting from its root directory to the last directory of the given pathname. Note that the term *path* here means the logical name resolution path (path of the pathname from one directory to another) and not the physical path of the network.

Figure 4 shows some typical examples of these three reliability factors. The broken lines in this figure indicate a message transfer for object accessing and not for name resolution. Hence, the broken lines are not taken into account for the name resolution reliability factors. Using the reliability parameters mentioned above, a Galaxy user can specify reliability requirements for the various (node, external name) pairs of interest.

Depending on the users' specifications, the system automatically replicates the necessary directories at the proper nodes. This approach seems logical because all the objects in a system are not of equal importance to all users and a particular user normally works at only a few nodes of a large distributed system. Using this approach, it is possible to give users the degree of reliability they want for resolving various names without greatly affecting the overall system efficiency.

Interprocess communication

The interprocess communication (IPC) facility has several uses in a system, including informing processes of asynchronous event occurrences, requesting processes to perform operations, and transferring data from one process to another. Thus, using a single IPC mechanism for handling all types of uses and for communicating among the processes at all different levels of the system structure will be very inefficient. This inefficiency relates to the amount of data to be transferred, the timing constraints, etc., which are normally different for all these cases.

Based on this observation, Galaxy provides different types of IPC facilities, giving users the flexibility to choose and use the most suitable facility for their specific application needs. Below, we discuss the various types of IPC mechanisms provided in Galaxy.

Interrupt message passing. A user explicitly wanting to avoid waiting for messages can trigger a software interrupt when the message is received. The interrupt service routine, executed in the context of that process, can then receive the incoming message and process it, or it can notify the main program of the event in a user-defined manner. A mechanism like this can allow processes that are inherently bound to computation to react quickly to incoming messages without using some form of message polling. It can also be used to

trigger language-defined exceptions across process boundaries.

Fixed-size message passing. Galaxy supports three types of communication facilities in this category:

(1) *Blocking type:* This type of facility can be used to implement IPC that, to the sender, looks like a procedure call. Just as a procedure caller passes on control to the procedure, so our sender yields to the receiver. The *send* request message passes the equivalent of procedure arguments, and the reply message returns the results.

(2) *Asynchronous type:* In this case, if the destination process is waiting for the message, the message is sent out; otherwise, control is returned immediately to the sender, and the request is recorded as an event. When the destination process receives the message, an interrupt to the sending process is executed. A similar scheme is used for an asynchronous receive.

(3) *Polling type:* Here, if the destination process is not ready to receive the message, control is immediately returned to the sender. In contrast to what happens in the asynchronous type, the request is not recorded; hence, there is no trace of the message. If the process "wants" to send the message later, the sender has to repeat the poll to ascertain whether the receiver is ready to receive the message. A similar scheme is used for a polling receive.

Message passing by memory sharing. Galaxy uses two memory-sharing approaches for IPC:

(1) *Virtual page transfer:* Here, when a page of data is sent from a source process to a destination process, the system modifies the page table of the source process to detach the page from the source process, and the page table of the destination process points to that page. The page data is neither copied nor moved to another place. The sender and the receiver should be synchronized for the page transfer. This mechanism is especially useful for transferring I/O buffers among the file servers, the device drivers, and the users.

(2) *Segment sharing:* In this method, data sharing is implemented by changing the mapping of virtual memory. After a source process sends a segment of data to a destination process, both processes can access the data. Two processes that want to share a memory segment must synchronize with each other.

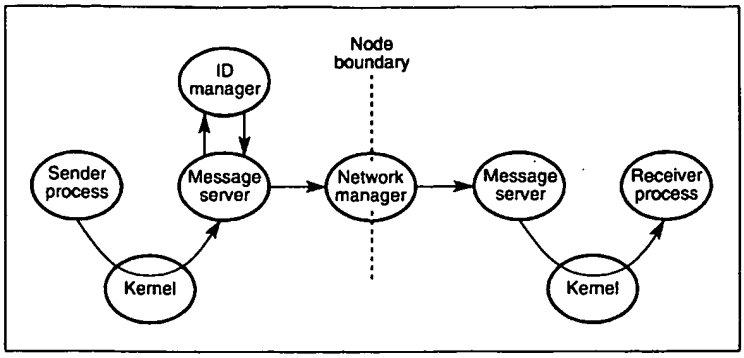

Figure 5. Network-transparent interprocess communication mechanism.

Networkwide IPC. In Galaxy, any IPC command finally causes a trap to enter kernel processing. The kernel searches the local process table for the receiver process of a communication. If the receiver is at the local node, it handles the communication locally; otherwise, the IPC request is forwarded to a special user-level module called the *message server*.

The message server locates the receiver by interacting with the ID manager and forwards the request message to the message server of the node on which the receiver resides. The request is serviced at the receiver's node, and the result is returned to the sender via the message servers of the two nodes.

Thus, when a sender transmits a message to a receiver, the sender has no direct means of determining whether the receiver is local to its node or is actually on a remote node. Hence, our IPC mechanism is network transparent, and the primitives for local as well as remote message passing are the same. Figure 5 illustrates our network-transparent IPC mechanism.

Computation model

Concurrent processing is one of the key concepts for achieving greater efficiency in the present computing era. The desirable properties of the process management mechanism of a system with concurrent processing include low overhead in process creation, deletion, and context-switching operations; flexible and efficient information sharing among the processes; and flexibility for users to define their own process scheduling policies.

In conventional operating systems, current solutions for these requirements are the concepts of light-weight processes and coroutines in process management.[10] However, because of their own limitations, neither light-weight processes nor coroutines are suitable for the needs of current and future parallel computing.

For example, the concept of light-weight activities has limited applicability for various reasons. These include the inability to support a large number of processes; the fairly large overhead involved in process creation, deletion, and context-switching operations; the lack of user flexibility to control scheduling and other aspects of processes; and the difficulty of distributing light-weight processes on the various nodes of a distributed system with no shared-memory facility.

Similarly, coroutines suffer from deficiencies such as lack of true parallelism, poor inter-coroutine communication facilities, lack of a mechanism to protect coroutines from each other if required, and the inability to distribute coroutines over different nodes of a distributed system. Furthermore, at present, concurrent processing capabilities provided by operating systems, distributed systems, and language systems are isolated and are not under a uniform interface. This makes it difficult to write programs that contain various types of processes and interactions among them.

Based on these observations, we realized that to fulfill the various application needs, an ideal process management mechanism should provide a variety of processes whose properties vary. We thus proposed the notion of *variable-weight processes* in Galaxy.

Weight is defined as the amount of resources owned and accessed by a process. The amount of the resources is measured in two dimensions: space and time. Variable-weight processes allow a program to use a suitable set of processes with different weights. Furthermore, a generic interface can be provided for a suitable weight of processes to be chosen at loading time or even at runtime.

Executor/domain model. To meet flexibility, efficiency, and parallelism computing requirements and support variable-weight processes, we have proposed a new computation model called *Executor/domain*. An executor is an active entity that either owns or has access rights to domains. A domain is a unit of information identifiable by its name.

For example, one or more pages of a file may form a domain. The set of domains either owned or accessed by an executor is called the *executing domain* of the executor. The set of domains common to two executors is said to be shared by the two executors. Since the executing domain is arbitrarily defined for each executor, the shared domains can be flexibly defined for a group of executors.

An executor itself is described by a data structure called an *executor descriptor*, which is also maintained in some domains. The descriptor specifies the detailed information (such as registers and stack area) for its execution. If desired, the user can select a bare CPU as the executor or choose a fully equipped process—a process with its address space and all other necessary resources. This structure is flexible and enables a user to choose variously weighted processes for job execution.

Executor scheduling is primarily a function of the operating system's kernel, but in some cases, enabling the user to write the scheduler is desirable. Galaxy allows user scheduling at the light-weight-process level. Most scheduling of light-weight processes is controlled by user-defined schedulers. The kernel supports only the operations of catching interrupt signals, access control, switching execution mode, etc., and this makes the kernel very light.

Hybrid approach for light-weight activities. Conventional operating systems take two approaches to implementing light-weight processes or threads: in-kernel implementation and out-of-kernel implementation.

In in-kernel implementation, the kernel performs scheduling; therefore, parallelism and preemption are given by the kernel scheduler. Potential additional cost is a serious concern with this approach. A trap or system call is necessary to cross the user process/kernel protection boundary when these operations are executed. In addition, the kernel must maintain the management data in its virtual address space.

On the other hand, coroutine packages are widely used in out-of-kernel implementation of light-weight processes. We discussed their disadvantages above. Moreover, making a coroutine preemptive requires extra cost.

To overcome these limitations, Galaxy uses a hybrid approach for implementing light-weight activities. Galaxy features two classes of light-weight executors:

(1) user-mode kernel-supported executors and

(2) user-mode executors in a single kernel-supported base executor.

One or more user-mode executors can be defined within each kernel executor. They share the whole address space of the kernel-supported base executor, which is defined by the domain to which the kernel executor is attached.

A kernel-supported user-mode executor, known as a *microprocess*, is an executor whose runtime context is maintained in the user address space; scheduling is performed in the user mode. However, in this class of executors, the kernel provides a clock-handling facility for creating preemptive executors. By sharing the address space with a kernel, the system call for the clock handling is executed with a small overhead.

In addition, the two classes of user-mode executors have the following special features:

(1) If a user-mode executor takes a page fault or another kind of trap, other user-mode executors within the same kernel executor can continue their execution.

(2) User-mode executors can issue distinct system calls and I/O requests.

(3) User-mode executors can be defined hierarchically; the scheduling is performed at each level of the user-mode executor hierarchy. This facility is especially useful for implementing hierarchical objects in an object-oriented environment.

Galaxy is still under development, and extensive performance comparisons with other systems have not yet been undertaken. But, our experience to date indicates that Galaxy's design is fundamentally sound, although some refinement and further efforts are necessary in a number of areas.

As of this writing, prototypes of major system components have been implemented. All but a few hundred lines of code are written in C, with the remaining lines written in assembly language. Galaxy runs on IBM RT/PC workstations.

The prototype implementations have demonstrated the viability and attractiveness of individual components. The full system is being implemented, and we are conducting experiments to measure the performance of each system component.

We plan to continue Galaxy research in the future in areas given low priority while we were setting our design goals. The areas include supporting a network of heterogeneous nodes, supporting resistance to maliciousness, and supporting real-time applications. ■

Acknowledgments

We thank the guest editors and anonymous referees for their valuable suggestions, which greatly improved the structure and presentation of this article.

References

1. G.J. Popek and B.J. Walker, *The Locus Distributed System Architecture*, MIT Press, Cambridge, Mass., 1985.

2. J.K. Ousterhout et al., "The Sprite Network Operating System," *Computer*, Vol. 21, No. 2, Feb. 1988, pp. 23-36.

3. D.R. Cheriton, "The V Distributed System," *Comm. ACM*, Vol. 31, No. 3, Mar. 1988, pp. 314-333.

4. D.R. Cheriton and T.P. Mann, "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance," *ACM Trans. Computer Systems*, Vol. 7, No. 2, May 1989, pp. 147-183.

5. B. Welch and J.K. Ousterhout, "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System," *Proc. Sixth Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 697, 1986, pp. 184-189.

6. X. Jia et al., "Highly Concurrent Directory Management in the Galaxy Distributed Operating System," *Proc. 10th Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 2048, 1990, pp. 416-423.

7. D. Ritchie and K. Thompson, "The Unix Time-Sharing System," *Comm. ACM*, Vol. 17, No. 7, July 1974, pp. 365-375.

8. R. Sandberg et al., "Design and Implementation of the Sun Network File System," *Proc. Usenix Conf.*, Usenix Assoc., 1985, pp. 119-130.

9. A.B. Sheltzer, R. Lindell, and G.J. Popek, "Name, Service, Locality, and Cache Design in a Distributed Operating System," *Proc. Sixth Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 697, 1986, pp. 515-522.

10. D.L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *Computer*, Vol. 23, No. 5, May 1990, pp. 35-43.

Pradeep K. Sinha is a researcher at the Tokyo Information and Communications Research Laboratory of Matsushita Electric Industrial Company of Japan. His research interests include distributed operating systems, distributed database systems, and programming methodology.

Sinha received his BE in computer science from Allahabad University, India; his MS in computer science and engineering from the Indian Institute of Technology, Madras, India; and his DSc in information science from the University of Tokyo. He is a member of the IEEE Computer Society and a life member of the Computer Society of India.

Mamoru Maekawa is an associate professor in the Department of Information Science, Faculty of Science, University of Tokyo, Japan. He wrote more than 50 papers that have been published in the areas of operating systems, performance evaluation, software engineering, database systems, and distributed processing systems.

He received both his MS degree (with distinction) and his PhD in computer science from the University of Minnesota. He is a member of the IEEE Computer Society.

Kentaro Shimizu is an associate professor in the Department of Computer Science and Information Mathematics at the University of Electrocommunications, Japan. His research interests include distributed operating systems and multimedia processing. From 1980 to 1985, he helped develop a Lisp machine.

Shimizu received his MS and DSc degrees in information science from the University of Tokyo. He is a member of the IEEE Computer Society.

Xiaohua Jia is a lecturer in the Department of Computer Science of the University of Queensland, Australia. His main research interests are distributed operating systems, and concurrency and consistency control of distributed systems.

Jia received his BS and MS degrees from the University of Science and Technology in China and his DSc degree from the University of Tokyo.

Readers may contact the authors at the Department of Information Science, Faculty of Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan.

Hyo Ashihara is working toward his DSc degree in the Department of Information Science, Faculty of Science, University of Tokyo, Japan. His research interests focus on distributed operating systems.

Ashihara received his BS and MS degrees in information science from the University of Tokyo.

Naoki Utsunomiya is on the research staff at the Central Research Laboratory of Hitachi in Tokyo. His research interests include high-speed communication protocols and distributed systems.

Utsunomiya received his BS and MS degrees in information science from the University of Tokyo. He is a member of the IEEE Computer Society and the Information Processing Society of Japan.

Kyu S. Park is a senior research scientist in the System Engineering Center of the Korean Institute of Science and Technology. His research interests include network communication protocols and distributed operating systems.

Park received his BE and ME degrees from the Kyungbook National University, Korea.

Hirohiko Nakano is a senior researcher at Hitachi, Ltd., Tokyo, Japan. His research interests include distributed operating systems and programming languages.

Nakano received his BS and MS degrees from studies in the Department of Information Science, University of Tokyo, Japan.

IEEE Xplore RELEASE 2.1

Home | Login | Logout | Access Information | Alerts | Sitemap | Help

Welcome United States Patent and Trademark Office

BROWSE SEARCH IEEE XPLORE GUIDE SUPPORT

e-mail printer friendly

AbstractPlus

View Search Results | Next Article

Access this document

Full Text: PDF (724 KB)

Download this citation

Choose Citation & Abstract

Download ASCII Text

» Learn More

Rights and Permissions

» Learn More

The Galaxy distributed operating system

Sinha, P.K. Maekawa, M. Shimizu, K. Jia, X. Ashihara, H. Utsunomiya, N. Park, K.S. Nakano, H.
Tokyo Univ., Japan;

This paper appears in: **Computer**
Publication Date: Aug. 1991
Volume: 24 , Issue: 8
On page(s): 34 - 41
ISSN: 0018-9162
CODEN: CPTRB4
INSPEC Accession Number:4002330
Digital Object Identifier: 10.1109/2.84875
Posted online: 2002-08-06 17:53:44.0

Abstract
The Galaxy research project, which is attempting to design, implement, and use a distributed computing environment, based on the idea of making gradual improvements by learning from existing systems and trying to overcome their limitations, is described. The design goals and novel aspects of the project are outlined, and the Galaxy system architecture is described. Galaxy's object **naming** and locating mechanisms, interprocess communication, and computation model are discussed

Index Terms
Inspec
Controlled Indexing
distributed processing network operating systems

Non-controlled Indexing
Galaxy distributed operating system Galaxy system architecture computation model distributed computing environment interprocess communication locating mechanisms object naming
Author Keywords
Not Available

References

No references available on IEEE Xplore.

Citing Documents

No citing documents available on IEEE Xplore.

View Search Results | Next Article

Help Contact Us Privacy & Security IEEE.org

Indexed by Inspec

© Copyright 2006 IEEE – All Rights Reserved

CS-TR-3671 UMIACS-TR-96-55

Using Content-Derived Names for Caching and Software Distribution

Jeffrey K. Hollingsworth
hollings@cs.umd.edu

Ethan L. Miller
elm@cs.umbc.edu

University of Maryland Institute for Advanced Computer Studies

Abstract

Maintaining replicated data in wide area information services such as the World Wide Web is a difficult problem. Ensuring that the correct versions of libraries and images are installed for application programs presents similar challenges. In this paper, we present a simple scheme to facilitate both of these tasks using content-derived names (CDNs). Content-based naming uses digital signatures to compute a name for an object based only on its content.

CDNs can be applied to several common problems of modern computer systems. Caching on the World Wide Web is simplified by allowing references to an object by its content rather than just its location. In a similar fashion, applications can request library objects by their content without having to rely on the presence of a file system hierarchy that the application recognizes. Further, applications that require different versions of an object can coexist peacefully on the same machine. While this idea is still in its early stages, we present experimental evidence from a study of World Wide Web objects that indicates that CDNs could reduce network traffic by allowing requests to be satisfied by differently-named duplicates with the same contents.

Keywords: digital signature, global object names, disambiguation, World-Wide Web caching, WWW object duplication

This page intentionally left blank (I've always *wanted* to say that!)

1 Introduction

The explosive growth of the World Wide Web (WWW) has placed ever-increasing demands on existing computer networks and hardware, and the introduction of WWW-centric languages such as Java has exacerbated the problem. Low bandwidth and high latency are only part of the problem, though. With the WWW linking millions of independent computers worldwide, a single object may be duplicated on hundreds of computers, each using a different local name for the object. Detecting these differently-named duplicates can reduce network bandwidth and free up cache space by allowing local servers to cache just one of the duplicates, satisfying requests for "different" objects with a single copy.

A related problem is the distribution of software. Each computer has its own file system hierarchy, often with different naming schemes that require per-site customization to get a package to work. Worse, software packages often rely on other libraries which themselves evolve. Someone using the latest version of one tool might require the latest version of a library, while another tool that hasn't been updated in several years uses an older version of the same library. Keeping track of different versions of software and matching them can be a daunting task made difficult by the need to customize each piece of software to take the local file system structure into account.

This paper addresses many of these concerns with a simple mechanism: providing a name for objects based solely on their content. This name is the same for any two objects with identical content, regardless of their location. However, it is different for two objects that have the same location-based name but, due to changes over time, different contents. Caching becomes much simpler because an object has the same name regardless of where it is stored; a request for an object could be satisfied if that object were already cached from a different location. Moreover, the software distribution problem is simplified. Since an object's name does not depend on the file system hierarchy, location-specific customization is minimized. If a program specifies a needed library with such a name, it is guaranteed to get the correct version of the library because different versions have different contents and thus different names.

We first describe the work on which we based our new idea for naming documents, including previous research on caching for the WWW and some of the theory underlying digital signatures. We then describe several situations where content-derived object names provide benefits, including WWW caching, and software maintenance. Next we present experimental evidence supporting our claim that a significant number of identical WWW objects with different names exist. We conclude with a look at the implications of using content-based names in URLs, and some possible directions for future work.

2 Background

While the idea of using content-derived names (CDNs) for document identification on the WWW is a new one, there has been substantial previous work in the areas of naming and caching on the WWW. This paper builds on that work, as well as previous work in digital signatures.

2.1 Naming in the World-Wide Web

The most wide-spread naming scheme in the WWW is the URL (Uniform Resource Locator) [1]. This scheme bases object names solely on their locations, requiring users to ask a specific server for an object. This method has several shortcomings, including the inability to specify an object that could be retrieved from any one of several servers.

The URN (Uniform Resource Name) [8] was introduced to address this problem. Instead of specifying the document's actual name, a client can use the URN to find a server that would return the actual name and location of the desired document. Using this method allows a single document to be served in several distinct locations, and provides a method for a client to find the document using a single name. However, some problems still remain. First, the central URN server must be involved in every request for the document, though this load could be lightened by caching URN to URL translations. A second, more important, issue is that of keeping track of replicas. It is straightforward to notify a central authority when a document such as a technical report is duplicated, since it occurs relatively rarely. For objects such as images, though, such duplication will occur far more frequently. As an object is used in more and more places, the URN servers for it will become increasingly loaded. Keeping a small number of central naming authorities for a document is not scalable. Even if a scheme is developed to allow URN servers to scale, anyone creating a copy must notify the original URN server of the duplicate.

Sollins and Masinter [8] mention the use an MD5 digital signature [7] inside a URN to uniquely identify a document. The MD5 signature was used solely to generate a unique object ID, and not to identify objects from different locations with the same signature. In this paper we show that recognizing hidden replication provides the potential for caching to reduce both server and network load.

One of the biggest problems created by the development of the WWW is the bandwidth and server load required to provide the objects request by WWW clients. However, in any caching scheme where objects are mutable, some mechanism must be provided to age or invalidate stale copies of objects.

Gwertzman and Seltzer [4] examined several approaches to insuring that requests do not return stale objects. The Alex and time-to-live (TTL) protocols both perform well, with TTL requiring less bandwidth but imposing a higher server load than Alex. However, both of these protocols are necessary because objects with the same name may not, in fact, be the same object. While including a CDN in an object's name removes the need for consistency protocols for static objects, some objects (10% according to [4]) are dynamically generated and would not benefit from the use of CDNs.

2.2 Digital Signatures

A key feature of CDNs is the use of a digital signature to assign a unique name to an object based on its content. Algorithms such as MD5 [7] are one-way functions that take an arbitrary sequence of bytes and produce a result that is likely to be different from that of any other input sequence. MD5 is well suited as the function to content derived names. The MD5 algorithm produces a 128 bit signature, and Rivest [7] claims that it is NP-hard to find another document with an identical

signature. Touch [9] has reported that it is possible to compute MD5 in software at the rate of over 10 MB/second on current RISC workstations; we feel this rate is more than adequate for our proposed use of MD5.

In order for a content-derived name system to work, it must probabilistically guarantee that two different objects will not share the same object identifier. A global information system such as the WWW, however, could have many billions of objects. Fortunately, the probability of such a failure is small. The probability that m numbers chosen randomly from a pool of n will be unique is $e^{-m(m-1)/2n}$ [5], where $n = 2^{128}$ for MD5. For 10^{15} objects, the probability of success (no two objects with different content have the same name) is $e^{-2^{-29}}$, assuming that object names are randomly distributed. Since $1 - e^{-x} \approx x$ for small x, the chance of failure is approximately 2^{-29}, or 10^{-9}. We believe this chance of failure is sufficiently low because it is below the probability that there would be an undetectable failure in a disk or network link somewhere in the world during that time. Thus, undetectable hardware failure is more likely to cause the use of an "incorrect" object than content-derived naming.

While 10^{15} objects would be sufficient for each of one hundred million servers to create ten million unique objects without name collision, 10^{15} unique objects may not be sufficient for many years of object creation by all of the computers connected to the WWW. However, there is no theoretical limit to the length of a digital signature. While MD5 only produces 128 bit signatures, a similar algorithm could be constructed to produce a signature with 256 bits, allowing the creation of 10^{30} unique objects with the chance of collision dropping to below 10^{-17}.This is sufficient to allow each of ten billion servers to create ten million unique objects *per second* for over three hundred years.

3 Applications of Content-Based Naming

Using content to name objects provides a simple way to request the specific instance of an object. It is extremely useful in situations where it is important that a specific version of an object be provided (such as software distribution). Likewise, it is also useful when it does not matter where the copy of an object is found (i.e., replication) as long as its the right object. Content-based names are the ultimate in location independent naming — they include no reference to what machine an object is stored on, nor do they require arbitrary filing hierarchies. In this section, we describe three uses for content-based naming: caching, software distribution, and world wide web search engines.

3.1 World Wide Web Caching

Perhaps the most obvious use for object names based on content is caching. Caching for the WWW has been studied in [2], in which several schemes for managing these caches were proposed. However, all of these caching mechanisms rely on the URL of an object to distinguish it from other objects.

This approach has two drawbacks. First, it is impossible to detect the existence of two identical files named by different URLs. In Section 4, we report that such files make up approximately 5% of all objects discovered by a WWW robot*. People who like images often include them in their own WWW pages, and will usually copy the images to their local server when they do so. Copy-

ing images reduces load on the original server, provides a safeguard in case the object is deleted from the original location, and reduces the amount of typing necessary to refer to the graphic in local WWW pages. While most of the objects are small, this situation is changing as small static graphics give way to miniature movies; as images grow in size, the bandwidth saved by caching at intermediate sites grows. Current WWW caching schemes do not recognize multiple instances of the same object if their names are different. URNs [8] attempt to solve this problem by providing a central name server to provide the name of the "closest" server, but are limited by the need to keep track of all of the copies of an object. Using an object's digital signature, on the other hand, has no such limitations. If two objects are the same, their digital signature will match. A WWW cache can look for objects bearing the same signature, and return any one that matches even if it was not from the specified server.

The second drawback to traditional caching approaches is that they are prone to caching stale data. Currently, this problem is solved in one of several ways. A WWW server may provide an estimated lifetime for a file; the cache in which the file is placed need not contact the server until this lifetime has expired. However, this approach has difficulty with objects whose lifetimes are highly variable, such as users' home pages. A single page may change several times in a day, or it may remain unchanged for weeks at a time. How can a cache cope with such unpredictability? Another consistency method requires the cache to ask the server when the object was last modified, and base its decision on the returned time and the modification time of the copy in the cache. This approach guarantees that the most up-to-date copy is used, but requires that the server handle a file stat operation for each object that is requested, even if the object is cached near the requester. A third method requires the server to "push" the new version of the object to machines that it believes have an old copy. However, this method introduces scaling issues.

If a digital signature is used, however, the cache can ensure that the object requested is the one it is holding by checking the signatures. If the signatures do not match, it must request a new copy from the object's server. The request goes to the file's server if and only if the file has changed; there are no spurious requests for files and all requests for a file are satisfied with the correct version.

Content-based naming depends on the relative immutability of objects on the WWW. If an object's contents change frequently while its location-based name remains the same, digital signatures will be of little use. Fortunately, the objects for which digital signatures will be of most use are least likely to experience small changes. Images and, eventually, multi-image objects are most likely to be copied around the WWW by users that want to include them in their WWW pages. These images are unlikely to be changed by individual users, and they are also the largest component of WWW traffic [2]. Keeping digital signatures for them will help to reduce the bandwidth requirements for home users who have plenty of disk space but are connected to the WWW via a slow link.

---

* This fraction would likely increase if the sample size is increased.

3.2 Software Distribution

The current trend in creating software is to re-use components such as classes, dynamic libraries, icons, and sound bites. For technical (size of the objects) or legal (different component vendors) reasons, different objects are stored as separate files. For a software product to work correctly, however, the different components must be compatible with each other. A single package may involve hundreds of individual files, each of which must be the right file in the right place in the directory structure. An incorrect version of a particular library or configuration file, or even the right file in the wrong place, can render the entire package useless. This situation is complicated by the evolution of software and the interdependence of software packages. A single computer often has dozens of software packages, some of which may require different versions of the same software library. Maintaining such systems is difficult at best, and installing new software is often a challenge.

The WWW has further complicated software configuration management by facilitating the distribution of software over the Internet. No longer is the installer a computer expert; instead, complex systems must be "installed" by less-experienced users. Languages such as Java [3] allow users to pull classes from many different locations, yet there is no guarantee that the files obtained in this way will actually work together. Some files may not work with the latest version of a Java class, instead requiring an older version. How can the software publisher specify a particular version of a Java class or similar object?

3.2.1 Ensuring Version Consistency

Currently, names, or names combined with a version string, are used to identify external components (such as dynamically linked library). However, using names as the basis of compatibility is problematic. Software quality assurance requires that product be tested with all compatible components prior to shipping. As a result, many products include copies of the tested components as part of their distribution. When the product is installed, the included components are installed on the target machine with the designated name. This ensures that the last installed product will have the correct components. However, any previously installed software may now break because it may rely on older (or newer) versions of components that have been replaced by more recently installed software.

Using digital signatures provides a good solution to finding consistent versions of objects. Each library, icon, or sound bite will have an object identifier computed using a digital signature. This signature uniquely identifies an object solely based on its content. We term this digital signature a content-derived name (CDN) since it can be used to fully specify a requested object. When an application (or a library) wants to reference or load an external component, it simply specifies the CDN for the desired object. A library (or perhaps the file system) then locates the requested object and loads it. Since different versions of the same software component will have different CDNs, each application will get the desired version. Consistency is also required for some Web pages. As web pages become more dependent on specific visual layouts, it is necessary to ensure that all components included on the page are the exact ones the author intended.

At software installation, each component included with the distribution is loaded only if its CDN is not already installed. On the other hand, a new version of a software package may leave

unmodified many of the files that it uses. These objects will retain the same object identifiers as in the older version of the software, allowing the user to load only the files that have changed since the last version was released. Objects can refer to other objects by their CDN forming a graph of object dependencies. The graph makes it possible to ensure that all required components for an application are installed.

Traditionally, shared objects have been distributed in relatively large units (libraries) because maintaining consistent versions was so difficult. However, the use of content-based naming allows the sharing of objects at a much finer granularity since the verification of consistent versions can be automated.

Sometimes it is possible for an application to be able to use more than one version. For example, an application might be compatible and have been tested with either of two similar object libraries. This case can easily be accommodated by lists of equivalent CDNs. As long as one of the objects specified in the equivalence list is present, the installation process does not need to load an object.

It is also possible that an application could be customized during installation or by the user at a latter time. For example, users might add custom macros to their word processing system. Customizations could be applied to either the application or to individual objects. However, customizations could potentially change the content of an object (and thus its CDN). To accommodate this situation, each object should contain a customization region that contains fields that can be changed. This part of the object would not be used in computing its CDN.

The overall structure of an object in this scheme is shown in Figure 1. An object consists of the object body, external object references, customization region, and its CDN. The object body contains the majority of the object, including its executable code. References to other objects or customization data are represented as pointers to the appropriate section of the object. Each object reference can be a list of CDNs for equivalent objects. Although this information is immutable, it needs to be in a designated section of an object so that the object manipulation routines can identify an object's external object references. The customization region has two sections. One to store customized references to other objects and the second to store free format data. The only requirement is that pointers from the object body can't be modified due to customization since this would change the object's CDN.

3.3 File Hierarchy Independence

A second benefit from digital signatures is file system location independence. Many packages require extensive per-site customization to tell the software where to find files it needs. However, these files usually have the same contents (except for information on the location of other necessary files!) regardless of where in the directory structure they are located. A package that refers to an object using its digital signature need only look it up in a database of signatures and objects. Objects can be assigned human-usable names; this will likely be necessary for other maintenance reasons.

A file system to store objects based on their CDN could be built. Such a file system would support efficient storage and linkage of CDN-based objects. In addition to objects referring to other

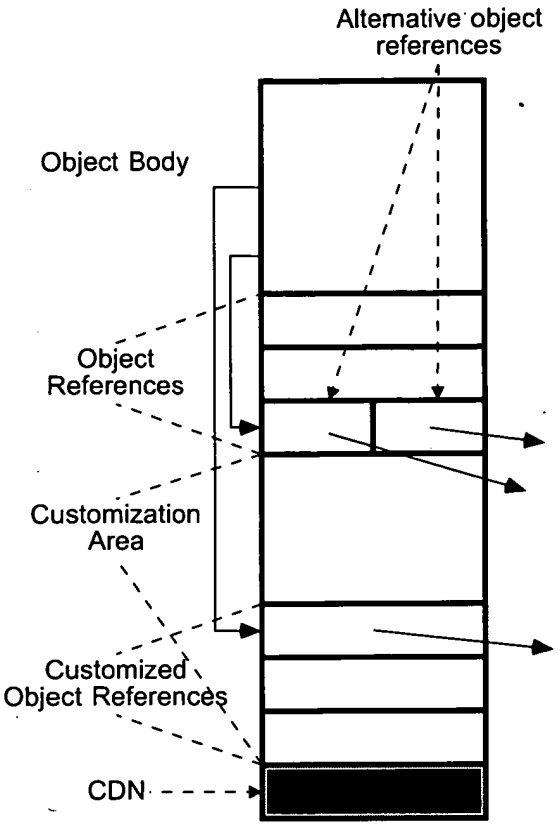

Figure 1. The structure of an object using content-derived names

objects, a user- visible namespace could be provided similar to the way a UNIX directory structure provides a user access to inodes. Objects would not be explicitly deleted from such a file system, but would be implicitly deallocated. Each stored object would have a reference count. When a reference count was decreased to zero, the object would be deleted. Due to the possibility of mutually referential objects creating unreachable cycles, a periodic garbage collection of the object will also be required. Replication in such a file system is easy since a request for a CDN can be serviced by any server that has the desired object. A request for a CDN is a request for specific data, and not a request to translate through a namespace. Traditional consistency concerns only apply for the user-visible name space, not the objects themselves.

In some ways, a CDN-based file system would be similar to the PILOT file system used on the Altos [6]. In the Pilot file system, all stored objects had a globally unique object identifier. However, this object identifier was created by concatenating the host identifier of the server where the object was created and a sever relative identifier. The Pilot file system also supported linking files together based on their object identifiers. However, a program using the file system needed to explicitly mark an object and immutable before its object identifier was frozen. With a CDN, no explicit designation is required. However, a server must explicitly make an object available, an operation similar in meaning to marking it immutable.

The ability to locate objects by an identifier based solely on their contents will simplify software distribution via the WWW. Already, programs such as Netscape use a single directory to cache retrieved objects. Currently, the identifiers for these objects in the cache bear no relation to the objects' contents. Using digital signatures instead would allow the "cache" to grow to the size of the entire disk. At that point, a Unix-like (or Mac-like, or Windows-like) directory structure could be used as an overlay, providing a user-friendly interface to a cache of objects fetched from the WWW. Any piece of software could be distributed in this way; the user could assign names to those pieces of software that she wished to access directly, such as the main executable for a word processing program. Other objects would be fetched as necessary. To allow disconnected operation, a package might arrive with a self-installer that contains all files necessary to run the program. Rather than install all of the files, though, such an installer would merely install the files not already present locally.

3.4 Web Search Engines

Content-derived names can also improve the quality and efficiency of web search engines. Many popular documents are replicated (mirrored) at several different sites. Each of these replicas will have different URLs, but the same content. Not only will the server name be different, but sites often arrange their file systems slightly differently so the path names of replicas will also be different. To the user of a search engine, though, each replica is identical. Unfortunately, most search engines currently generate different "hits" for each of these copies. However, if the search engine maintains a CDN for each object, duplicate hits due to replication can be presented as a single hit with several choices. Merging replicas can save time for users of search engines and reduce load on servers since users will only visit a single site. Use of CDNs by search engines can easily be added since it only requires changes to the internal information of search engines, not any web protocol.

CDNs can also be used to assist in the maintenance of search engines indices. To see if a document has changed, search engines must either fetch and re-index the document or query a server to discover the modification time of an object. While using modification time provides an indication if an object has changed, using a CDN such as MD5 would provides a better indication if a document has really changed.

In addition, each server could export a list of publicly visible objects stored on that server and their CDNs. This list could be periodically updated by the server. Search engine crawlers would then be able to compare the URL to CDN mappings provided by the server to previous values. Out of date index entries for document would be identified by differences in their CDNs.

In addition, mapping information can also be used by a crawler to identify if a URL is a replica of some other object that has already been indexed. If the object (as indicated by its CDN) is a copy of an existing object, the crawler would not need to request the document from the server. Instead, it would simply add an entry for the new copy. This method works equally well for identifying when a document has not changed, but has simply be moved to a new part of the server's file system. Again, the URL will be different, but the CDN will remain the same.

4 Experiments

To evaluate the amount of object replication in the web, we wrote a simple web robot to scan objects and record their URL and MD5 hash values. The idea was to try to find pages on the web that have different URLs, but are in fact the same object. Since we were interested general object duplication, we computed MD5 values for all objects found including HTML documents, images, and Postscript files. We term the same object with two (or more) different URLs as a name aliased object.

[NOTE TO REVIEWERS: This results in this section are very preliminary. A more detailed study will be presented in the final paper. It will include a larger number of pages, starting from several randomly selected URLs, and a more complete analysis of the results.]

The scanning program was written in Perl and configured to scan recursively to a depth limit of 13. The robot was started at a single web page, and left to run. The robot scanned and computed the MD5 hash value for 28,974 web pages*. After running this robot, we then looked for different URLs that resulted in the same MD5 hash value. We were interested in finding out not only how many URLs mapped to the same object, but also finding out why the same object was being replicated.

Because the scan was done using a robot, rather than measuring traffic requests, the data presented is based on a static count of WWW pages rather than on dynamic access counts, such as those available by monitoring traffic through a proxy cache). Dynamic data would be more useful for investigating the potential opportunity for caches to identify replicated objects since it captures temporal re-use of objects. Static data has the advantage that it is easier to gather information from disparate parts of the web since proxy based monitoring would be biased by idiosyncrasies in the proxy's usage pattern.

Type	Count	Percent
fragment identifiers (# characters)	3,130	10.8
host alias	298	1.0
substring match	427	1.5
name aliased object, same server	471	1.6
host alias and name alias	58	0.0
Other matches	430	1.5
Total duplicates	4,814	16.6

Table 1. Duplicated objects by type of replication.

A summary of duplicate objects is shown in Table 1. Of the 28,974 URLs scanned, there were 24,156 different MD5 hash values. The majority, 3,130 (65%) of the duplications were due different fragment identifiers that were part of the URL. Fragment identifiers (preceded by an octothorpe) are used to identify views, often relative offsets, into documents. These are generally uninteresting duplications because it is trivial to eliminate them. Another source of duplication

* Due to time constraints, the search was terminated before a complete scan at this depth was finished.

was host name aliases. This type of duplication occurs when a single host has several DNS names that map to the same IP address. This type of duplication was responsible for 298 (6%) of the duplicates. Again, this type of replication could be detected by existing name based schemes if the names are defined based on IP address rather than host name.

The remaining four categories of duplication, totaling 1,386 objects (29%), would be impossible to detect using name based object equivalence. Three of these four categories are due to intra-server aliasing of object names. The category "substring match" is the case where one URL name is a substring of another URL name. Substring aliases were responsible for 427 duplications or 9% of the duplicates. Substring aliasing results from servers mapping requests for a directory names (e.g., /foo/bar) to a specific pages (e.g., /foo/bar/index.html). It is easy to visually identify these aliases. However, since the mapping from directory name to default page can be configured on a per-server basis, it is impossible for caches or robots to identify these aliases based on name. Same server name aliasing of objects also occurs where one name is not a substrings of the other. This results from either having duplicate copies of the same object stored on a server, or from local file system aliasing (e.g. symbolic links). This type of aliasing was responsible for 471 duplications (10%). The final type of intra-server aliasing was a combination of host name (DNS) alias and local file system alias. This was relatively rare and accounted for only 58 of the duplications seen.

The final category of name aliasing is duplication of the same object on different servers. We found 430 duplicates (9%) were due to inter-server name aliasing. This type of duplication results from replication or cloning of objects. We would like to distinguish replication, an explicit copy for availability or load balancing, from cloning, a copy of a useful object (such as a red ball). The distinction between replication and cloning is important because several proposals have been made to accommodate replication in the web and other distributed information services. However, all of these schemes require explicit management of replication and do not accommodate cloning. Unfortunately, it is difficult the accurately identify if copies are replicas or clones. However, for content-based naming, the fact it is the same object with two different names is all that is important.

Type	All Objects		Duplicated Objects	
	Count	Percent	Count	Percent
text/html	23,654	81.6	1,099	79.3
image/gif	3,622	12.5	253	18.3
text/plain	959	3.3	16	1.2
image/jpeg	422	1.5	11	0.8
Other	316	1.1	7	0.5
Total	28,973		1,386	

Table 2. Object types.

We also investigated the question of whether different object types were more likely to be replicated than other types. A comparison of the types of all objects with those that were replicated is shown in Table 2. In both cases, the most common object type is a HTML document (approximately 80% of all documents scanned were HTML). The fraction of replicated objects that are gif

images was somewhat higher than the rate of occurrence of gif images in the overall sample space. This matches the intuition that people "borrow" GIF images from other web pages to use them on their pages.

5 Future Directions

The work presented in this paper is still in early form, and we see several different ways in which this idea could be used to build better systems. One of the difficulties with the use of CDNs as part of an object's WWW name is that changes of this scope require wide agreement to be effective. However, CDNs could still be used on a small scale to test their effectiveness. The HTTP protocol allows user-defined fields to be transmitted along with an actual document; one of these fields could contain the MD5 digital signature of the document. The client or proxy cache could cancel the transfer if it already had a document with that digital signature, possibly saving bandwidth by removing the need to transfer the entire object. This would be particularly useful for larger images since smaller documents, less than a few kilobytes, would be transferred in their entirety before the message to stop would arrive at the source. Nonetheless, we believe this strategy would prove particularly useful in caching image files and binary files — many sites mirror software distributions, yet detecting that the files are the same even though they come from different sites is difficult.

Another immediate use for CDNs is in WWW search engines. Currently, search engines are unable to detect duplicate documents stored in different places. Our experience with several searches using Digital Equipment's Alta Vista search engine (http:/altavista.digital.com) turned up many instances of the same document being found on different sites. This problems occurs both for WWW pages and Usenet articles that are archived at several sites around the Internet. Simply keeping MD5 digital signatures for each page would allow the search engines to return shorter lists, and allow a user a choice of site from which to fetch a document that she wants.

Mobile computing can also benefit from CDNs. One of the problems in mobile computing is the ability to operate a computer away from its "base," since requests for objects must eventually be sent through the network to the mobile computer's home file server. CDNs present an attractive alternative: a mobile computer can merely ask the local file servers for an object using its CDN. The mobile computer need not follow the same file pathname conventions as the local server, since the object is identified solely by its content. Moreover, the mobile computer can check that it received the object it requested by computing the digital signature on the object, so it need not even trust the local server.

Eventually, if network-based software distribution replaces physical distribution, most of the disk space on client computers could be turned into a cache for network objects. When a commercial software package is purchased, a request for an object would fetch it from the WWW if it was not already cached. There would be no need for garbage collection; an object would simply be removed from the cache to make room for new objects. If that discarded object were needed again, it could be refetched. Much research on caching strategies and other issues is necessary before this goal can become a reality, but its implementation would greatly simplify the operation of computers in an environment where access to the WWW is constant and omnipresent.

6 Conclusions

In this paper we have presented a new approach to naming objects for distributed systems and software configuration. Rather than using user-assigned names to identify objects, we proposed to derive object names automatically based on the content of an object. Our scheme makes it possible to identify references to the same object even if the objects have completely different names. We described how content-based naming of objects can be used to manage cache and replication in a distributed information service, to ensure consistent software configuration, and to improve the performance and quality of WWW search engines.

Content-based naming represents an attractive alternative to creating location-independent names for objects. By splitting the namespace from the content-space, it is possible to decouple namespace issues — hierarchical vs. relational naming, name aliasing from content-space issues such as caching, replication, and storage management.

Finally, we presented the results of a robot that scanned for duplicated WWW objects with different names. We discovered that approximately 5% of the objects found were duplicates of other objects, supporting our claim that using CDNs will provide immediate benefit to proxy caches and search engines in addition to the longer-term benefits to software distribution and WWW caching.

References

1. T. Berners-Lee, L. Masinter, and M. McCahill, Uniform Resource Locators (URL), RFC 1738, Network Working Group, 1994.
2. A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, "A Hierarchical Internet Object Cache", USENIX — Winter 1996 Conference. January 1996, San Diego, CA, pp. 153-163.
3. J. Gosling, B. Joy, and G. Steele, The Java Language Specification. 1996: Addison-Wesley.
4. J. Gwertzman and M. Seltzer, "World-Wide Web Cache Consistency", USENIX — Winter 1996 Conference. January 1996, pp. 141-151.
5. R. Motwani and P. Raghavan, Randomized Algorithms. 1995: Cambridge University Press.
6. D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell, "Pilot: An Operating System for a Personal Computer", Communications of the ACM, Feb 1980. **23**(2), pp. 81-92.
7. R. L. Rivest, The MD5 Message-Digest Algorithm, RFC 1321, Network Working Group, 1992.
8. K. Sollins and L. Masinter, Functional Requirements for Uniform Resource Names, RFC 1737, Network Working Group, 1994.
9. J. D. Touch, "Performance Analysis of MD5", SIGCOMM. Aug 1995, Cambridge, MA, pp. 77-86.

IP.com PriorArtDatabase

Technical Disclosure

Archived Kernel Extensions

Author(s): **IBM TDB**
Mealey, BG
Swanberg, RC

IP.com number: **IPCOM000123901D**
Original Publication Date: **June 1, 1999**
Original Disclosure Information: **RD v42 n422 06-99 article 422131**
IP.com Electronic Publication: **April 5, 2005**

IP.com, Inc. is the world's leader in defensive publications. The largest and most innovative companies publish their technical disclosures into the IP.com Prior Art Database. Disclosures can be published in any language, and they are searchable in those languages online. Unique identifiers indicate documents containing chemical structures as well as publications open for comment in the IP Discussion Forum. Disclosures are published every day online and also appear in the printed IP.com Journal. The IP.com Prior Art Database is freely available to search by patent examiners throughout the world.

Client may copy any content obtained through the site for Client's individual, noncommercial internal use only. Client agrees not to otherwise copy, change, upload, transmit, sell, publish, commercially exploit, modify, create derivative works or distribute any content available through the site.

Note: This is a pdf rendering of the actual disclosure. To access the notarized disclosure package containing an exact copy of the publication in its original format as well as any attached files, please download the full document from the IP.com Prior Art Database at: http://www.ip.com/pubview/IPCOM000123901D

ip.com

www.ip.com

Copyright © IP.com, Inc. All rights reserved.

Archived Kernel Extensions

Disclosed is a method for managing multiple versions of a kernel extension or device driver for a single system. The method employs existing technologies utilized in archival of loadable shared objects, applied at the kernel level for loadable kernel images.

Supporting multiple versions of the same kernel extension or device driver in a single installed system image presents several difficulties to installation, system administration, and update procedures. For example, with the evolution of today's hardware systems, 64-bit technology has required operating systems to now provide 64-bit kernels. The 64-bit kernels have in turn required kernel extensions and device drivers to then become 64-bit, all the while continuing to support their 32-bit predecessors. A further complication is the ability for some processors, such as the *PowerPC, to run either a 32-bit or 64-bit environment. This gives the customer the flexibility to choose to run a 32-bit operating system kernel or a 64-bit operating system kernel depending on specific scalability and application workload demands. With this flexibility comes the requirement that at least two versions of every kernel extension and device driver be available for selection dependent on the operating environment chosen.

Using traditional kernel extension and device driver packaging and configuration methods, this would have required another complete set of kernel extension and device driver installable file sets to be created and maintained. Unique extension names would have been introduced to distinguish the 64-bit versions from the 32-bit versions, also impacting all areas where an extension or driver is known by name. Much of the content of these file sets, with the exception of the extension binaries themselves would be identical, but duplicated. Also, when switching between 32-bit and 64-bit operating environments, somehow the appropriate set of kernel extensions and device drivers would need to be activated over the other.

The disclosed solution solves each of these problems by maintaining a single name space per kernel extension and device driver. The solution is to package each kernel extension and device driver in an archived file format which contains multiple versions of the extension or driver. Specifically in the case of 64-bit versus 32-bit operating environments, the archived file would contain the 64-bit extension binary and the 32-bit extension binary. The kernel loader is then enhanced to recognize the archived file format when loading the extension, perform a run-time check to determine the current operating environment and then conditionally load the correct kernel extension member from the archive for the selected environment. This allows all entities referencing the extension or driver by name to remain unchanged. Thus major functions like system installation, system administration, service update, and switching between 32-bit and 64-bit operating environments are not impacted by the existence of multiple kernel extension or device driver versions present on the system.

*PowerPC is a trademark of International Business Machines Corp.

Disclosed by International Business Machines Corporation

Software Deployment on Network Storage Based Systems

Todd Poynor
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2001-257
October 12th , 2001*

software deployment, software installation, diskless, network storage

This report briefly summarizes a number of recent investigations related to software deployment on network storage based systems that our team recently performed. A specific context in which this was investigated was for a pool of general-purpose servers in a single data center that are to be allocated among different applications and/or different customers according to demand or level of service purchased. Separating software installations from server machines increases the modularity of the data center, such that the appropriate level of compute power can be more easily provisioned for a specific software environment and the available compute power can be more efficiently allocated among software environments. A number of additional capabilities and concerns unique to "diskless" software deployment are discussed, as are general topics in improving the modularity and flexibility by which software packages are installed, configured, combined into software stacks, and shared among servers in cluster environments.

* Internal Accession Date Only
Approved for External Publication
© Copyright Hewlett-Packard Company 2001

1. Introduction

This report briefly summarizes a number of recent investigations related to software deployment on network storage based systems that our team performed as part of the Distributed Service Utility research project at HP Labs. This information is being published to the research and development community in order to document some of the ideas, results, and ongoing projects that sprang from that effort.

Our use of the term *network storage based systems* refers to architectures where storage resources may be dynamically paired with compute, I/O, and other resources that make up a general-purpose server system – that is, the server has no local disk. The term encompasses both the so-called Network-Attached Storage (NAS) and Storage Area Network (SAN) architectures and associated protocols.

Here we are concerned with those portions of the software lifecycle called *software deployment*, and in particular these activities in the breakdown of that process by Carzaniga *et al* [13]:

- ***Install***: initial insertion of a system into a consumer site.
- ***Activate***: starting up the executable components of a system.
- ***Deactivate***: shutting down any executing components of an installed system.
- ***Update***: a special case of installation that can often rely on the fact that many of the needed resources have already been obtained during the installation process.
- ***Adapt***: modifying a previously installed software system, initiated by such events as a change in the environment of the consumer site.
- ***Deinstall***: removal of a system from a consumer site.

The activities from their breakdown that are not included are *release* and *derelease* (*retire*), which are activities of software producers. We also considered including configuration change tracking for troubleshooting (as a part of fault management or security management), such as to track software revisions installed, or to record when a patch was installed or a configuration parameter was modified.

2. Motivation and Design Principles

Here software deployment is investigated in the context of systems comprised of, among other things, numerous general-purpose servers in a single data center. The servers are to be allocated among different applications and/or different customers (in the case of a service provider's data center hosting multiple customers) according to demand or level of service purchased. Diskless servers for executing applications and other software are employed in service of these goals:

- **Efficient redeployment of compute power and applications** through separation between software installations and processing resources, allowing these two components to be associated in modular fashion such that the set of servers running a software installation may vary dynamically (and cost-effectively) according to need. Each change in association is to occur relatively quickly due to: the "stateless" nature of the servers; leveraging the previous work to install software from media and, to some degree, configure the software; and improvements in the process by which software is automatically configured to run on new servers and in dynamic environments.

- **Decreased hardware costs** due to lack of per-server disks and even I/O busses.

We use the term "software environment" for a collection of OS and application software, plus associated control information, housed on network storage resources that corresponds to a traditional installation of OS and application software on a local disk. The installation of a software environment occurs not to install the software on a particular server, but in order to make the environment available to run on any number of appropriate servers. We wish to quickly redeploy servers to execute different software environments according to need, as in response to changing workloads or customer or resource sets. By separating software environments from servers, we can accomplish this in a more dynamic (and cost-effective) fashion than when software environments and servers are tightly bound together. The appropriate level of compute power can be more easily provisioned for software environments, and the available compute power can be more efficiently allocated among competing environments.

Such an architecture has the above advantages, and others noted below, over other projects, such as IBM Océano [14], that are also aimed at dynamically allocating data center resources but that employ traditional directly-attached disks. Other research projects that make use of network storage for efficiency of server reallocation include Muse at Duke University [17].

The same techniques may be used not just for general-purpose servers based on such operating systems as UNIX and Windows but also for reprogrammable networking infrastructure functions running on multi-purpose platforms.

3. Sharing and Replicating Software Environments

There are well-known benefits to sharing software environments and/or individual software products among multiple servers, and even across multiple customers hosted at the same data center. Sharable software packages also lend themselves to efficient replication on multiple machines. This can be greatly beneficial in manually constructing a cluster of identical servers and in automated deployment of additional instances of software (as to automatically meet increased demand). Sharing installations helps speed the time to construct or replicate an environment, helps reduce the amount of disk space consumed by duplicate static files, and offers advantages in centralized management of environments over the deployment life cycle, such as to reduce the need for duplicate actions applied to multiple environments. Although these concepts also apply to traditional systems with local disks, for network storage based systems these capabilities are vital to deriving the full benefit of the architecture.

Most tools for replicating installations are geared toward automating the steps necessary to perform the installation of a cluster of identical servers with local disks that are expected to stay in that configuration for some time. For our purposes we want to quickly and possibly automatically redeploy servers to execute differing software environments according to need in highly dynamic fashion.

For effective shared deployment of software, important considerations include whether software subscribes to a file system layout policy that enables cross-system sharing of software and whether the installation procedures are organized into one-time *install* and per-replication *configure* phases, as described in the subsections that follow.

3.1 File System Layout Policy

The policies by which OS and application files are placed in the file system hierarchy play a large part in the way software is shared. This is influenced by such factors as:

1. Is there a distinction between the directories to which sharable, read-only files are installed and the directories in which administrator-modified configuration files and programmatically-written log files and other per-instance files are installed or created?

2. Is there a clear separation between the directories to which separate software products install sharable files, such that there is a clear mapping between an OS or application product and the directories needed to obtain access to the product (and only that product)? Preferably, each product installs all sharable files beneath a single product root directory.

For example, the HP-UX file system hierarchy is organized (à la System V.4) into sharable vs. host-private mount points [3], with structure for separating optional applications from the base OS, to ease and speed replication of software in NFS-mounted clusters. Linux software has traditionally been less consistent in adherence to such a file system layout policy. In fact, many separate packages install into common directories **/usr/local/bin**, **/usr/local/lib**, **/usr/local/man**, etc., in direct conflict with point #2 above. There is a move toward greater organization of the layout in such Linux distributions as Red Hat, which adopts the Filesystem Hierarchy Standard (FHS) [12].

3.2 Per-Server Configuration

Many of the options for efficiently sharing and replicating software configurations boil down to separating the installation task into a one-time *install* action and a per-target *configuration* action. In this fashion, read-only bits may be shared among multiple servers, as via NFS mount points, and host-private information may be configured per server using such means as package-specific per-server configuration performed by scripts. For example, the HP-UX installation utility SD-UX subscribes to this model [4, 5].

On systems that either do not subscribe to amenable file system layouts and/or do not implement a per-server configure phase in the installation process, it is possible compensate for this to some degree using automation. We can "wrap" the existing tools to help automatically factor out installation and configuration steps and installation directories, or we can modify the tools and software packages to add explicit *install* and *configure* phases. Methods of automatically determining *install* vs. *configure* actions include determining "deltas" of file information as in the NT **SysDiff** [10] and HP-UX **mkpkg** [11] utilities, and instrumenting the file system protocols (designs for which are still under investigation and may be described in a future report). Potential future directions include enhancing existing installation tools and system software to enable these features.

Of key interest has been investigation into methods of improving the flexibility of software configuration, in part so that even software with highly complex interactions with other software and other network resources may be efficiently replicated to new machines. Although not necessarily specific to network-storage-based systems, we briefly note some of these here. Improved languages for expressing dynamic portions of configuration files using markup languages such as XML have been one area of investigation. Another area that has been pursued is a framework for software to dynamically configure and respond to changes in the surrounding environment, such as to establish and break ties with other software and hardware services, as needed over the lifetime of execution of the program [15]. Some additional investigation that was performed looked at management information models by which software and servers may be described in machine-readable formats for automation of deployment (such as expressing dependencies on other software and hardware platforms), expressing service level requirements, and so forth. This effort looked for improvements over existing mechanisms such as the DMTF CIM Application Management Model [18], which has limited support for distributed systems

[20] due in part to its origins in traditional network operations management [21], and the Marimba/Microsoft Open Software Description (OSD) [19], which has a number of limitations noted in [22]. The general topic of dynamically reconfigurable and replicable software remains of high interest as part of the utility computing programs at HP Labs [16], and work continues in these areas.

3.3 Avoiding Maintenance of Local Disks

A number of products exist for replicating an installation onto multiple servers with local disks. For HP-UX, Ignite-UX is used for this purpose; on Linux the most popular tools include VA SystemImager [1], REMBO [2] (which also replicates NT and other Windows OS'es), Red Hat KickStart [6, 7] (based on RPM [9] packages), and IBM Linux Utility for cluster Installs (LUI) [8].

Part of the operation of these tools is to load remote software packages or file system images onto a local disk at boot time. For Network storage based systems, downloading of files to local disks is replaced by NFS mounting of remote file systems at a considerable increase in initial speed of redeployment.

Reallocating a physical disk device from one customer to another exposes the danger that data belonging to the previously allocating customer remains on the disk for potential access by the newly allocating customer. For this reason, it is normally necessary to reformat the disk, wiping out all data stored previously, when reallocating a disk to a new customer, a rather time-intensive operation (the IBM Océano project [14], for example, does this). Network storage based systems can avoid the need for reformatting physical devices by controlling access to most storage resources, providing only file-system-level interfaces to files created by the customer and not providing general disk-block-level access to these resources.

3.4 Management of Software Environments

A number of management tasks are needed for the individual software environments, such as to browse, delete, etc. the environments, as well as the "catalog" of available environments. If environments consist mainly of file system images residing within the file system, existing file system management tools may provide a minimum of functionality, but may not be sufficient to implement an easily managed system, especially for a very large number of environments. There are also a number of additional actions that may be useful for shared software environments, such as to independently upgrade a particular software package or select a subset of servers that are to be upgraded, and to modify the set of servers that are to execute a certain environment. Potential future activities were to address this topic.

3.5 Security Implications

Data centers shared among multiple customers have certain special security considerations, including protection against unauthorized access to the software deployed. Common file system access controls probably suffice for protection of file system images against snooping or corruption by other than the owning customer. It may, however, be advantageous to allow instantiation of the same "base" file system image for multiple customers, and even the details of what software a node is running may expose more information than one would like.

Different customers may not want to share read-only file system mount points due to such concerns as security, performance, and deviations from usual practices of accounting for disk space used. When a

customer creates a software environment that includes a software package used by multiple customers, even sharable portions of its file system image could be copied from a protected location to a new copy for that customer only, in order to alleviate this concern. There is little new security guarantee provided, however, since shared access to an uncorrupted original image is still assumed, unless access is only granted to a trusted party higher up in the service provider chain (such as the data center owner).

4. Automated Software Environment Assemblage

The modularity principles that separate software environments from compute power can be extended to increase modularity of the software packages that comprise an environment. Improvements in, or automation of, the processes by which different software packages are assembled into a software environment, upgraded or deleted in multiple software environments at a time, etc. could provide substantial improvements in manageability. This can also play a part in satisfying goals of modularity between software environments and compute power by automating the process by which a software environment is assembled from components appropriate for a particular server's hardware.

Rather than requiring an administrator to manually install and configure each software environment, we can provide tools and automation to help construct software environments, such as to assemble together Red Hat Linux 6.2 plus an Apache Web server plus the appropriate files that comprise the Web content for a customer (as we prototyped). This could enable such activities as:

- Combining software packages and data on the fly to match changing demand. For example, a "hot" web site could be moved from a shared Web server system to a dedicated system, or the appropriate OS for a particular server could be combined with OS-independent Web site content as required by the server architecture.
- Performing version control, such as to backtrack to a known good configuration after an installation that goes awry.
- Upgrading software simultaneously for several environments. For example, the Apache Web server software could be upgraded to a new release for each software environment that includes the Apache Web server.

One approach is to base the solution on installation tools and software packaging formats that separate the tasks of installation and configuration, using techniques as previously employed for diskless clusters (and for single-point administration clusters). A variant of this idea is to write new scripts for existing software packages that encode knowledge of how to access NFS mount points for software shares and to configure combinations of software. Another approach, which may be employed to some degree with unmodified toolsets and software packaging, records the effects of an initial install for later replay in another environment or perhaps for undoing the installation. Existing models include NT **SysDiff** [10] and HP-UX **mkpkg** [11]. An area of potential further investigation concerns how these deltas may be manipulated to achieve automatic assembly of environments.

5. Conclusion

Modular server farm architectures based on network storage have clear advantages in quick and efficient allocation of server resources dynamically matched with software environments according to policy. This report has provided some background on this design principle and has described a number of

directions that have been considered, including a number of areas that we continue to be pursue, for improving the deployment and lifecycle management of software in such an environment.

Acknowledgements

Many of the architectural principles on which this report is based were originally espoused by Lance Russell, Bert Munoz, and Tung Nguyen of HP Labs. The ideas and positions on the application of these principles to software deployment documented here were formulated by a team at HP Labs led by Tom Wylegala, the membership of which also included Steve Hoyle, David A. Barrett, Dan Conway, Ilan Ginzburg, and Baila Ndiaye.

References

[1] VA SystemImager documentation at http://systemimager.sourceforge.net/.

[2] REMBO documentation at http://www.rembo.com/docs/rembo/.

[3] *HP-UX 10.0 File System Layout White Paper* at **/usr/share/doc/file_sys.ps** on an HP-UX 10.0 or above system.

[4] *NFS Diskless Concepts and Administration* white paper at **/usr/share/doc/NFSD_Concepts_Admin.ps** on an HP-UX 10.01 or above system.

[5] *File Sharing and Other Helpful Facts for HP-UX 10.0 Developers* white paper at **/usr/share/doc/dev_apps.ps** on any HP-UX 10.01 or above system.

[6] JWCS information on Red Hat KickStart at http://www.redhat.com/mirrors/LDP/HOWTO/KickStart-HOWTO.html.

[7] Red Hat Linux 6.2 Reference Guide section on KickStart at http://www.redhat.com/support/manuals/RHL-6.2-Manual/ref-guide/s1-kickstart2-howuse.html.

[8] IBM LUI information at http://oss.software.ibm.com/developerworks/opensource/linux/projects/lui/.

[9] RPM home at http://www.rpm.org/.

[10] *12 Steps to Cloning Windows NT Systems with SYSDIFF.EXE* at http://www.winntmag.com/Articles/Index.cfm?ArticleID=1.

[11] Staelin, Carl. *Mkpkg - A Software Packaging Tool*, HPL Technical Report HPL-97-125R1 at http://lib.hpl.hp.com/techpubs/97/HPL-97-125R1.html.

[12] Filesystem Hierarchy Standard at http://www.pathname.com/fhs/.

[13] Carzaniga, Antonio, *et al. A Characterization Framework for Software Deployment Technologies*, University of Colorado Technical Report CU-CS-857-98, 1998 at http://www.cs.colorado.edu/users/alw/doc/AvailablePubs.html.

[14] Appleby, K., Fakhouri, S., *et al. Océano – SLA Based Management of a Computing Utility.* Proceedings IFIP/IEEE International Symposium on Integrated Network Management 2001.

[15] Poynor, Todd. *Automating Infrastructure Composition for Internet Services.* To appear at the 15th USENIX Large Installation System Administration (LISA) Conference, December 2001. Also published internally at HP as HP Labs Tech Report HPL-2001-212.

[16] Wilkes, John, Goldsack, Patrick, *et al. Eos – The Dawn of the Resource Economy.* Proceedings HotOS VIII, May 2001.

[17] Chase, Jeffrey B., Anderson, Darrell C., et al. *Managing Energy and Server Resources for a Hosting Center*. Proceedings 18th Symposium on Operating System Principles (SOSP), October 2001.

[18] DMTF *Understanding the Application Management Model* at http://www.dmtf.org/spec/Whitepapers/CIM_Applications_wp.pdf.

[19] OSD -- Describing Software Packages on the Internet at http://www.marimba.com/products/whitepapers/osd-wp.html

[20] CIM Tutorial: Applications and Namespaces at http://www.dmtf.org/educ/tutorials/cim/extend/apps.html.

[21] Aschemann, Gerd and Kehr, Roger. *Towards a Requirements-based Information Model for Configuration Management*, ICCDS IV, 1998 at http://gemini.iti.informatik.tu-darmstadt.de/~kehr/publications/iccds98.pdf.

[22] Hall, Richard S., *et al. Software Deployment Languages and Schema.* University of Colorado Technical Report CU-SERL-203-97, 1997 at http://www.cs.colorado.edu/users/rickhall/deployment/SchemaPaper/Schema.html.

Security for Automated, Distributed Configuration Management

P. Devanbu, M. Gertz, S. Stubblebine*
Contact: devanbu,gertz@cs.ucdavis.edu
stu@cs.columbia.edu
April 25, 1999

Abstract

Installation, configuration, and administration of desktop software is a non-trivial process. Even a simple application can have numerous dependencies on hardware, device drivers, operating system versions, dynamically linked libraries, and even on other applications. These dependencies can cause surprising failures during the normal process of installations, updates and re-configurations. Diagnosing and resolving such failures involves detailed knowledge of the hardware and software installed in the machine, configuration manifests of particular applications, version incompatibilities, etc. This is usually too hard for end-users, and even for technical support personnel, specially in small businesses. It may be necessary to involve software vendors and outside consultants or laboratories. Employees working on sensitive, proprietary projects may even have to resort to calling the help line of an application vendor and discussing details of their desktop configuration. In order establish valid licensing, the user may be forced to disclose additional details such as the user's identity, machine identification, software serial number, etc. This type of disclosure may reveal proprietary information or (worse) security vulnerabilities, and increase the risk of attack by hackers or cyber-criminals. An adequate solution to the distributed configuration management problem needs to address the security concerns of users, administrators, software vendors and outside consultants: keeping details of installations private, authenticating licensed users and software vendors, protecting the integrity of software, secure delegation across administrative boundaries, and protecting proprietary information. Existing commercial and research systems [12, 8] provide distributed configuration management by distributing configuration information and software over local and wide-area networks. They provide flexible, automated, distributed configuration management. However, many or most of the security issues listed above remain to be addressed. These issues are the central focus of our research.

*Address for the first two authors: Dept. of Computer Science, Room 2063, Engineering Unit II, Davis, CA 95616. Last author: CertCo Inc., 55 Broad St., Suite 22, New York City, NY 10004

1 Introduction

Traditionally, the installation of software was simple: an administrator received a tape, unbundled the software off the tape, created a few initialization and profile files, and the software was off and running. Later releases for the software would arrive likewise in magnetic media and be handled likewise. With component-based distributed software running on a networked computer, the process is simpler in some ways and more complex in others. Rather than using cumbersome physical media, software can be dispatched over the network (perhaps even automatically as agents or applets). However, there may be several constituent components, from different vendors. The configuration and installation process for each of these may be distinct, and there may be a nontrivial integration step. In addition, each networked computer may have different resources and user requirements, and may require customized software installation.

Configuration management continues to be an issue after initial installation: the configuration of a machine may change, for example, due to the installation of a new device driver or an upgrade to the operating system. A vendor may produce new releases of a software product to fix faults or enhance functionality, thus leading to the installation of new files that may have consequences beyond the specific product itself.

Papers from the University of Colorado Software Engineering Research Laboratory (UC SERL) (See, for example [9]) have described a useful characterization of the *software deployment lifecycle* from the perspective of the software producer, and the software consumer:

- From the software producers point of view, there are two significant lifecyle events: *release*, when the software (or a new version) is first introduced, and *retire*, when an old version is deprecated by the vendor.
- From the software consumers point of view, there are four significant[1]:
  1. *Install:* the first installation of the software product.
  2. *Reconfigure:* Selects a new configuration (from the list available) of an installed software in response to new requirements at the consumer's site. E.g., the user may require some a dictionary for a new language in a word processor.
  3. *Adapt:* A change made in the consumer's configuration requires a change to the installation of a product. For example, a new printer may be installed, which requires the installation of new fonts.
  4. *Update:* A change made to a software product in response to a change initiated by the software vendor.

We adopt the UC SERL view of the software deployment lifecycle. SERL has developed an architecture (the Software Dock [8]) to address the needs of this lifecycle model. Ar-

[1]We present here only a simplified view of the deployment lifecycle. See [9] for full details.

chitectures for configuration management have also been devised by Marimba [12] and the Desktop Management Task Force consortium [2].

Our goal in this paper is to consider the security problems that arise in software configuration management.

2 Security Issues in Distributed Configuration Management

The most common situation is a distributed computer network where individual machines have complex installations tuned to the needs of specific users. In this context, there are various security goals, such as privacy, authentication, and delegation etc. We illustrate the need for these security features with an example.

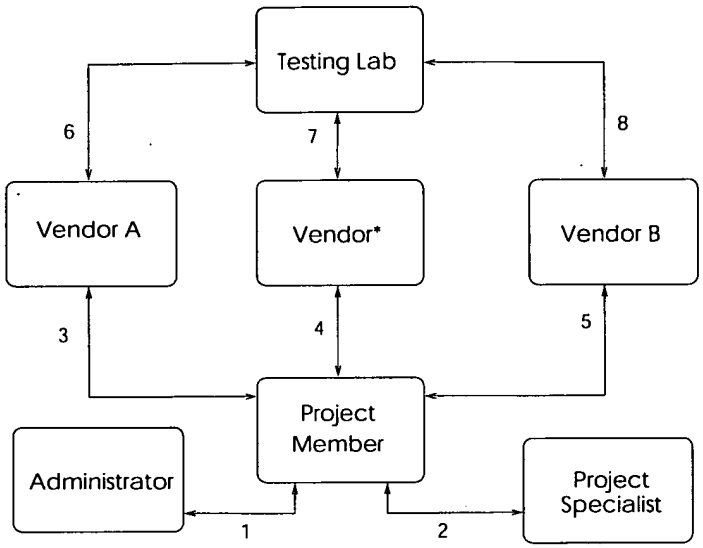

Figure 1: An Example Distributed Configuration Management Scenario

2.1 An Example

Consider Figure 1. A project member signs up to work on a special project, and checks with the departmental network administrator (edge 1) to find out what software is needed. The overworked administrator tells the project member to consult with a project specialist (edge 2) (an outside consultant) for details on the software required. The administrator in a sense here is "certifying" the project specialist as the responsible person for configuring software for the project.

The project specialist identifies an application $A*$ and vendor for the project member, who contacts the vendor (4) to obtain the product. The vendor (Vendor*) uses components C_A, C_B from two other vendors (Vendor A) and (Vendor B); C_A and C_B occasionally have conflicts. To determine if such conflicts exist, the Vendor* always consults a testing lab (7) which obtains versions (6,8) and tests them (or has already tested them in the past). The lab informs Vendor* (7) of the compatible versions. The project member finds the right version information from Vendor* (4) and obtains the right versions of the components (3,5).

We use the term "testing lab" advisedly; in practice, information such as this is obtained by calling the tech-support line for Vendor*, which results in much wasted time, often without any positive results. Another, more ad-hoc approach is to comb the web and usenet groups. This process is unpredictable, time-consuming, and often also fruitless. Another (more expensive) approach is to hire a knowledgeable consultant, who plays the role of a "testing lab" and makes it his/her business to be aware of incompatibilities. We envision a configuration framework that supports this more directly, and thus use the term "testing lab" illustrate a more systematic approach, which allows organizations with such capabilities to offer these services for a (possibly electronically delivered) fee. This type of dialog with vendors of different components and testing labs could be repeated for all the different applications (both work-related and personal) that the project member installs.

This scenario just covers the *install* part of the consumer side of the deployment lifecycle. Now consider a *reconfigure* event. The project member decides to install a video camera device for teleconferencing; in the process of installing the device driver, a new component is loaded, which causes the installed application from Vendor* (described above) to fail. The project member contacts the project specialist, who after much investigation, fails to resolve the problem, and refers the project member to Vendor*. The project member contacts Vendor* (4)'s support service; and identifies herself as a licensed user of $A*$. Vendor*, after obtaining an elaborate description of the problem, and the project member's installation, realizes that the driver uses a newer version of C_B that conflicts with the version of C_A that is currently installed. Vendor* then contacts the testing lab, which identifies a newer version of C_A that is compatible with the newer version of C_B. Again, as before, this situation is somewhat idealized; in practice, the project member (or any user) spends many hours trying to localize the problem, with or without the help of a voice at the other end of a support line; more often than not, Vendor* is unable or unwiling to diagnose the problem. The situation remains often unresolved, and the user is forced to abandon either the effort to reconfigure his system, or the use of application $A*$.

A similar scenario can be described for an *update* whereby a new version of $A*$ is released. Vendor* may inform the project member about a new update of $A*$; if this new version is authorized by the project specialist the project member may go ahead and obtain the update from $Vendor*$. This update may have unintended effects on other software that the project member has installed, which in turn may trigger additional dialogs between the project member, vendors of other software, their testing labs etc. These dialogs will involve description of hardware/software installed on the project member's machine, proof of licenses

being held etc.

Many applications ship with a range of configuration options that can be adjusted to suit the needs of a specific user's hardware/software set up on a specific machine. Thus, without adjusting the rest of the installation and contacting other vendors etc., it may be quite feasible to adjust the configuration/installation of an application to get it perform satisfactorily. However, the tuning of these options, specially for complex applications such as business process support, databases, and even word processing (with special needs for fonts, print drivers etc) can be quite complex, and require skills and resources beyond the capabilities of the systems administrator of a small organization. So even with flexible applications, the best solution in often to simply outsource the configuration task to an external entity.

We hasten to emphasize that we do not claim to advance a solution for the configuration management problem *per se*. Incompatibilities and other difficulties such as those defined above may happen, and will often require human effort to diagnose and solve them. Our goal rather is to ensure that *if such information is available*, it is promptly, reliably and efficiently delivered to those who need it, subject to certain security requirements that we shall discuss briefly now, and in more detail later.

2.2 Security Issues in this example

There are several security issues that arise here. First, the project member ends up revealing a great deal of information about the configuration of her machine to the project specialist, vendors, and testing labs. Certainly, there is a strong threat to the privacy of the project member (*e.g.*, the project member may have personal applications or software on the machine, and my unwittingly reveal this information by simply disclosing the existence of a particular DLL or component). In addition, detailed knowledge of the configuration of the project member's machine is made available to a number of outsiders, who may be now be able to attack known vulnerabilities in the machine, and gain proprietary secrets or engage in cyber-crime or cyber-terrorism.

Second, there are a number of authentication issues. The vendor needs to know that the project member is a licensed user of the relevant software. In addition, the project member needs to know that versions of the software that are installed have not been tampered with and modified, and are actually the right versions prescribed by the project specialist, the vendors, the testings labs etc.

Finally, there are delegation issues. The administrator delegates configuration authority to the project specialist, and the project specialist in turn delegates some authority to vendors and testing labs. These delegations may have associated time-periods of validity. These delegations need to be handled in a secure and timely fashion.

These issues are discussed in greater detail in Section 4.

3 Current Approaches

Architectures such as Marimba [12] and the Colorado Software Dock [8] deal explicitly with this problem. All architectures consist of these elements

1. A **language** for describing sites, configurations and updates. This language has facilities for *explicit* hierarchical descriptions of configurations, listing the various required elements for a software installation, as well as implicit *constraints* on software (such as requirements for certain types of functionality, without explicitly identifying the component).
2. An **Event Messaging** mechanism, for notifying events relating to the deployment lifecycle to vendors, customers, etc.
3. An **agency** on the customer side and on the vendor side for making use of the configuration descriptions, site descriptions, and the event notifications to derive consistent configurations and download the necessary software.

While the different approaches [2, 12, 8] have different advantages and disadvantages (See [1] for a survey), we primarily concerned here with security issues. In the following section, we identify the security issues that are of concern in the networked configuration management context.

4 Research Issues

We identify several critical security needs in systems that perform distributed software configuration. Some of these are handled by existing systems; others are not adequately dealt with. We are currently developing a flexible, retargetable architecture that addresses the security needs; this paper lays out the requirements and issues that must be addressed by such an architecture.

Integrity is the property that a data item (software, data, etc) is intact, and has not been tampered with. In the context of software configuration, integrity requirements arise in different contexts:

Software Integrity Software that is shipped from the vendor must arrive intact at the installation site. For example software from Vendor* that arrives to the student's PC (link 4, Figure 1) must be checked for integrity and completeness.

Configuration Integrity The configuration of a machine at a user's site must not be tampered with by unauthorized personnel.

Message Integrity Messages describing configurations (both correct ones and inoperable ones) must arrive correctly at the needed sites.

Cryptographically, integrity is established using message authentication codes [13] (MACs) or signatures. These techniques can be used in this context. Current systems such as Marimba implement this requirement via digital signatures.

Authentication It is often necessary to establish the authenticity of a message or a data item, *e.g.*, to make sure that the message really did originate where it claims to have.

Authenticating Software Vendors We need to verify that the originator or source of the software is indeed who it is claimed to me. For example, in figure 1 we need to establish that the delivered component C_A in step 3 really did originate with Vendor A and was not subject to tampering.

Authenticating Software Users During the *reconfigure* or *update* scenarios, when a customer contacts a vendor for help, the vendor needs to establish that the caller is actually a licensed user.

Cryptographically, authentication is established using signatures [13] within a public-key system. Other issues arise in this context, such as securely associating an entity or a role with a public-key, which is dealt with below under "delegation".

Privacy refers to the goal of keeping certain information secret. There are different types of privacy goals in the software configuration context.

Privacy of Content Software configuration activities involve the exchange of several valuable pieces of intellectual property, some of which may have commercial implications. Thus, certainly software vendors may want to encrypt the software prior to shipping it to paying customers. It may also be desirable to insert watermarks into software binaries to identify the origin of illegal copies. Testing labs or consultants obtain configuration rules (such as incompatibility of C_A with certain versions of C_B) at great expense, and may wish to keep this information private, and only reveal it to paying customers.

Privacy of Configurations Currently, users seeking to diagnose configuration problems are forced to reveal not only their identities (to establish proper licensing) but also details about the configuration on their site. In general, a user may not wish to reveal this information. Consider the situation where an employee of Microsoft is forced to reveal information about the configuration of her PC to a Lotus Technical support person to help diagnose a configuration problem. In the context of automated configuration management engine, this information would be asked for and transmitted without user intervention, thus leading to the risk of unwanted and unknown disclosures.

There is only one known technique for inquiring about subscription information without revealing the identity of the inquirer yet insuring that the entity doing the inquiring is authorized to access this information. It is called, Unlinkable Serial Transactions (UST) [16]. However, UST must be used in conjunction with some form of network anonymity

mechanisms such as the Anonymizer[17]. [2] Although UST doesn't inherently enable the server to profile the client. Application data can very well do this. For example, two queries having the same unusual subsets of components are likely to be queries from the same client.

Secure Delegation aims to selectively enable certain entities to perform certain actions. For example, delegating certain entities to take on certain roles [15] such as *administrator* or *specalist* or $Project_X$ $Configurator$ or *Testing Lab*. The scenarios described above illustrate different types of delegation. The student has some limited authority of his workstation. Also the system administrator has other authority. Each can delegate authority they possess to others. The student may delegate to the support staff the authority to a) add files to user space that are needed for the application at hand, and 2) to distribute some information about the user space configuration to an entity trusted by the system administrator . This is one type of delegation.

The support staff in turn delegates software configuration for one specific course to the teaching assistant; this amounts to "delegation of delegation". In another type of delegation, V* delegates to the testing lab to find out which version of C_A is compatible with which version of C_B. This amounts to delegation not of where to obtain software, but where to obtain configuration *rules*. In a fully automated configuration management architecture, these delegations, need to be authenticated through the use of certificates (some elements of the needed functionality have been described in [5])

Marimba [12] allows certain limited types of delegation through the use of *channels* and *sub-channels*. However, none of the systems allow fully certified delegations, delegations of delegations, delegations of configuration management rules etc.

5 Research Plans

We are interested in exploring several key issues that would underly a *secure* architecture for distributed software configuration over the internet. In this section, we give a list of issues that arise in this context, and are central to our research.

Languages Automatic configuration management hinges on an expressive description language. Current configuration management languages (CML) [11, 18, 2] are adequate for describing manifests and configuration rules. But they completely ignore security requirements such as authentication, delegation, etc. Our goal is to introduce such security features into CMLs. We take an approach to CMLs as being based on an object-oriented data model (OODM), as used in object-oriented database systems [14].

[2] Other techniques such as the Anonymizer [17] do not, alone, address the issue of whether the end user is authorized to query the end server. This is why UST is needed.

Modeling configuration, querying configurations and messaging among communications will all be based on this OODM. Security features such as delegation and privacy, can be implemented as view definitions on the configuration data. Constraints on valid configurations can be expressed as constraints on the data. The data, constraints and the view definitions are certified and protected using public-key cryptography. Determining the correct configuration at a site (for installation or for update) will be implemented as a query evaluation procedure, which collects data from the available views and attempts to find a satisficing answer. This will be implemented by extending the set-based cryptographic certificate distribution techniques described in [7] to an object-oriented model.

Cryptographic Techniques underly many of the goals described in the previous section. Some currently available techniques are quite relevant to the *privacy* and *authentication* goals outlined above (for example, UST [16] and Anonymizer [17] for protecting the privacy of the user while providing authentication to the software vendor). Our goal is to adapt these techniques for use in software configuration management. For the *delegation* goals described above, we will use the view definition approach outlined above, where the views are defined using cryptographically signed certificates [7].

Messaging Infrastructure Current approaches (See Section 3) all include a messaging infrastructure. Marimba [12] favours a "push" model; the software dock [8] has a hybrid model. In a situation where security goals are given a high priority, the use of either "push" or "pull" carries risk. For example, a "pull" system may give rise to unwanted disclosures: the query processing entity being "pulled" with configuration-related queries may be able to chain queries together and derive information about the querying entity. By the same token, "push" models may unintentionally reveal valuable information to unauthorized entities if the "push" channels are not carefully managed and connected. In our view of configuration derivation as query evaluation, the problem becomes one of optimizing the distributed evaluation of a database query over a database distributed over several sites, subject to security constraints, where certain sites may not have access to certain data.

Formal Underpinnings Configuration management is key to proper functioning and security of a system, and can thus be viewed as part of the critical infrastructure of an organization. In this context, we believe that formal verification of the correctness of a configuration management approach is well worth the costs. We are interested in developing formal underpinnings of our approach. Important properties to establish include:

Correctness : derived configurations at a site do not violate any rules or manifests as provided for within the applicable delegations and authentications.

Completeness : derived configurations have all *required* elements as per application delegations.

Minimality : derived configurations do not have *unnecessary* elements or versions thereof.

Timeliness : derived configurations are updated as soon as needed information is available from applicable delegations.

Security : Applicable goals of privacy (items that should be kept secret are available only to individuals who are allowed to know about them) and authentication (entities that are legitimate licensees are allowed to perform actions allowable to licensees) are met [10].

Retargetability A key architectural concern in our work is the level of effort required to integrate our approach with existing approaches to configuration management [11, 12, 8, 2] (that do not consider security to the same level that we do). We are interested in generative [3] or object-oriented [6, 4] approaches to retargetability.

6 Conclusion

In this paper, we have described the difficult security issues that arise in distributed software configuration management, using an example. Existing systems solve some of these problems; many issues remain. The goal of our research effort is to develop a retargetable, customizable security framework that can be crafted on to existing configuration management architectures.

References

[1] Reidar Conradi and Bernahrd Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2), June 1998.

[2] Desktop Management Task Force. *Software Standard Groups Definition, Version 2.0*, Mar 1996. `http://www.dmtf.org/tech/apps.html`.

[3] P. Devanbu. Genoa - a customizable, front-end retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology, (accepted, to appear)*, 1999.

[4] P. Devanbu, R. Chen, E. Gansner, H. Muller, and A. Martin. Chime: Customizable hyperlink insertion and maintenance engine for software engineering environments. In *International Conference on Software Enginering (to appear)*, 1999.

[5] P. Devanbu, P.W. Fong, and S. Stubblebine. Techniques for trusted software engineering. In *Proceedings of the 20th International Conference on Software Engineering*, 1998.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[7] Carl Gunter and Trevor Jim. Policy-directed certificate retrieval, 1999. `http://www.cis.upenn.edu/papers/qcm.ps.gz`.

[8] Richard S. Hall, Dennis Heimbigner, Andre van der Hoek, and Alexander L. Wolf. An architecture for post-development configuration management in a wide-area network. In *17th International Conference on Distributed Computing Systems*, May 1997.

[9] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A cooperative approach to support software deployment using the software dock. In *International Conference on Software Enginering*, May 1999.

[10] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems. *ACM Transactions on Computer Systems*, 10(4), 1997.

[11] MARIMBA, MICROSOFT, TIVOLI, and NOVELL. OSD: Overview of the Open Software Description Standard, 1998. `http://www.microsoft.com/workshop/delivery/download-/overview/osd_overview.asp`.

[12] MARIMBA, INC. Castanet product family, 1998. `http://www.marimba.com/datasheets/-castanet-3_0-ds.html`.

[13] Alfred J. Menezes, Paul C. van Oorschot, Scott, and A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[14] OBJECT DATABASE MANAGEMENT GROUP (ODMG). *Object Database Standard ODMG 2.0.* Morgan-Kaufmann, 1997.

[15] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, February 1996.

[16] P. Syverson, S. Stubblebine, and D. Goldschlag. Unlinkable serial transactions. In *Financial Cryptography*, volume 1318 of *Lecture Notes in Computer Science.* Springer-Verlag, 1997.

[17] The ANONYMIZER website. `http://www.anonymizer.com`.

[18] Andreas Zeller and Gregor Snelting. Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology*, July 1997.

DERWENT-ACC-NO: 2004-711278

DERWENT-WEEK: 200648

COPYRIGHT 2007 DERWENT INFORMATION LTD

TITLE: Active networked peripheral device drive **files** maintaining method in networked computing system, involves inserting driver **files in file** system sub-directory to which **unique** identifier **name** comprising version identification is **assigned**

INVENTOR: ANTONOV, E G; HORAL, M P ; LYTLE, T J ; ORLETH, R E ; ROTH, T ; TING,
A L

PATENT-ASSIGNEE: MICROSOFT CORP[MICT]

PRIORITY-DATA: 2003US-0403788 (March 31, 2003)

PATENT-FAMILY:

PUB-NO	PUB-DATE	LANGUAGE	PAGES	MAIN-IPC
EP 1465065 A2	October 6, 2004	E	034	G06F 009/445
JP 2004303252 A	October 28, 2004	N/A	038	G06F 013/10
US 20040215754 A1	October 28, 2004	N/A	000	G06F 015/173
CN 1534449 A	October 6, 2004	N/A	000	G06F 003/12
=K R200408675 7A	April 10, 2020	0	300	G06F F013/10

DESIGNATED-STATES: AL AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HU IE IT LI LT
LU LV MC MK NL PL PT RO SE SI SK TR

APPLICATION-DATA:

PUB-NO	APPL-DESCRIPTOR	APPL-NO	APPL-DATE
EP 1465065A2	N/A	2004EP-0006871	March 22, 2004
JP2004303252A	N/A	2004JP-0108212	March 31, 2004
US20040215754A1	N/A	2003US-0403788	March 31, 2003
CN 1534449A	N/A	2004CN-1003232	March 26, 2004

INT-CL (IPC): G06F003/12, G06F009/445 , G06F013/00 , G06F013/10 ,

G06F015/173 , H04L012/24

ABSTRACTED-PUB-NO: EP 1465065A

BASIC-ABSTRACT:

NOVELTY - The set of driver files constituting a version of a networked peripheral device driver component is inserted into the established file system sub-directory. An **unique** identifier **name** comprising identification of version including date, encoded, multisegment value is **assigned** to the sub-directory.

DETAILED DESCRIPTION - INDEPENDENT CLAIMS are also included for the following:

(1) system for maintaining active networked peripheral device driver;

(2) multi-level active network device driver storage/access framework;

(3) method for maintaining multiple versions of **driver packages** in network; and

(4) computer-readable medium storing program for maintaining components of active networked peripheral device driver.

USE - For maintaining driver files of active networked peripheral device such as printers, scanner, multi-function peripherals, in networked computing system.

ADVANTAGE - **Unique** identification of driver **files** facilitates preventing a version of a driver **file** from overwriting other version of driver **file** bearing the same **name,** when a subsequent version of a **files** is loaded on a networked system's active **driver storage which ensures completeness of packages**.

DESCRIPTION OF DRAWING(S) - The figure shows the block diagram of the network
peripheral device arrangement.

CHOSEN-DRAWING: Dwg.5b/10

TITLE-TERMS: ACTIVE PERIPHERAL DEVICE DRIVE **FILE** MAINTAIN METHOD COMPUTATION
SYSTEM INSERT DRIVE **FILE FILE** SYSTEM SUB DIRECTORY

UNIQUE IDENTIFY
NAME COMPRISE VERSION IDENTIFY **ASSIGN**

DERWENT-CLASS: T01

EPI-CODES: T01-F01B; T01-F05B; T01-H05A; T01-S03;

SECONDARY-ACC-NO:
Non-CPI Secondary Accession Numbers: N2004-563930

Microsoft Windows

Versioning in the Windows Driver Foundation

September 13, 2006

Abstract

This paper provides information about versioning support in the Microsoft® Windows® Driver Foundation (WDF) for the Windows family of operating systems. It describes the versioning policy and presents scenarios that show how versioning applies in common situations.

This information applies for the following operating systems:

Microsoft Windows Vista™
Microsoft Windows Server® 2003
Microsoft Windows XP
Microsoft Windows 2000

The current version of this paper is maintained on the Web at:

http://www.microsoft.com/whdc/driver/wdf/WDFVersioning.mspx

References and resources discussed here are listed at the end of this paper.

Contents

Introduction 3
Versioning Policy 3
  Major Releases 3
  Minor Releases 3
Framework Distribution 4
  KMDF Contents 4
  UMDF Contents 5
Framework and Driver Installation 5
Binding to the Framework 6
  Dynamic Binding for KMDF 6
  Binding for UMDF 6
Versioning Scenarios 7
Best Practices for Vendors 8
Resources 9

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2006 Microsoft Corporation. All rights reserved.

Microsoft, Windows, Windows Server, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

© 2006 Microsoft Corporation. All rights reserved.

Introduction

The Microsoft® Windows® Driver Foundation (WDF) supports versioning for both the user-mode driver framework (UMDF) and kernel-mode driver framework (KMDF). Versioning enables two major versions of a framework to run side by side and provides for updates of minor versions. A vendor's installation package indicates the correct WDF version. At installation, the Microsoft-supplied co-installers check the target system for the correct version and update it if necessary.

Hardware and driver vendors should understand the versioning policy, know how WDF is distributed and installed, and be aware of the effects of versioning during common hardware and driver installation scenarios.

Versioning Policy

Every WDF release has a major version number and a minor version number. For example, the first release of KMDF was version 1.0. The next release was the minor version 1.1. A driver can run with a release that has the same major version number and a minor version number that is greater than or equal to the version of the library that it was built against. Thus a driver built against KMDF version 1.0 can run with KMDF 1.1, but not with KMDF 2.0. A driver built against KMDF version 1.1 can run with KMDF 1.1 but not with KMDF 1.0 or 2.0.

The versioning policy describes how Microsoft releases new versions of WDF and what happens to existing installations when a user installs a new version. The same versioning policy applies to both KMDF and UMDF.

Major Releases

A WDF release is assigned a new major version number if it includes any of the following:

- Incompatible changes to the device-driver interface (DDI)
- Many significant new features
- Changes that require the recompilation of driver source code

WDF supports side-by-side installation for two major releases. This means that a single system can have both a framework library for the current major release and one for the previous major release in use at the same time.

Microsoft intends to release major versions to software developers with the Windows Driver Kit (WDK) and in general distribution releases (GDRs). End users receive major releases with operating system releases and on Windows Update.

Minor Releases

A WDF release is assigned a new minor version number if it does not meet the criteria for a major release. In general, a minor release includes only the following:

- Bug fixes
- Security fixes
- New features that do not require the recompilation of driver source code

A minor release can contain either critical or noncritical fixes (or both). Every minor release is cumulative; that is; it contains all the changes that have been made in all

© 2006 Microsoft Corporation. All rights reserved.

the previous minor releases. Consequently, a user who has version 1.1 can upgrade directly to version 1.3; installation of version 1.2 is not required.

Microsoft intends to release minor versions to software developers with the WDK. In addition, such releases are currently available on the WHDC Web site. If a minor version includes a critical security update, Microsoft will also release it on Windows Update. End users control the download of such updates by using the Automatic Updates application in Control Panel. By default, critical security updates are automatically copied to the user's machine.

Each new minor release overwrites the previous minor release. After installation of a new minor release, all existing WDF drivers that require the same major version bind to the new minor version. Two minor versions of the same major version cannot run side by side.

Framework Distribution

Microsoft distributes each WDF framework in two ways:

- As a native product that is supplied with Windows
- As a redistributable package

The native versions of both KMDF and UMDF are supplied with Microsoft Windows Vista™ and later releases. Updates to the native version are distributed through Windows Update as described in the previous section.

The redistributable WDF package is a co-installer that hardware and driver vendors receive in the WDK and in GDRs. Vendors supply the redistributable WDF package as part of their installation package. The native WDF is provided as a resource in the co-installer dynamic-link library (DLL). The co-installer is a signed component. Driver installation fails if the certificate with which the co-installer was signed is not available on the target system.

To install WDF, an end user must have adequate privilege to install new software, according to the rules applied by the Add or Remove Programs application in Control Panel.

KMDF Contents

The native WDF for KMDF consists of two files:

- WdfLdr.sys, which loads the driver framework library
- Wdf*MM000*.sys, where *MM* is the major version number, which contains the framework library

In Windows Vista, the native KMDF is provided with the operating system. For Microsoft Windows 2000, Microsoft Windows XP, and Microsoft Windows Server® 2003, the native version is available in the WDK and on the WHDC Web site. If the native KMDF requires critical security fixes, Microsoft will release the updated version on Windows Update.

The redistributable KMDF is provided in a co-installer file that is named WdfCoInstaller*MMmmm*.DLL, where *MM* is the major version number and *mmm* is the minor version number. This file includes both the loader and framework library as resources. The redistributable co-installer is not supplied with the operating system or on Windows Update. Vendors acquire it in the WDK and in GDRs.

© 2006 Microsoft Corporation. All rights reserved.

The following paper was originally published in the

Proceedings of LISA-NT:
The 2nd Large Installation System Administration of Windows NT Conference

Seattle, Washington, USA, July 16–17, 1999

STATE-DRIVEN SOFTWARE INSTALLATION FOR WINDOWS NT

Martin Sjölin

© 1999 by The USENIX Association
All Rights Reserved
For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer. Permission is granted for noncommercial reproduction of the work for educational or research purposes. This copyright notice must be included in the reproduced paper.
USENIX acknowledges all trademarks herein.

State Driven Software Installation for Windows NT

Martin Sjölin
*Warburg Dillon Read (WDR)**
P.O. Box, 8098 Zürich, Switzerland
martin.sjoelin@wdr.com

Abstract

We have implemented a state driven installation mechanism to simplify the installation of software under Windows NT. We have a "central configuration database" which defines the target state of the machines, e.g. a declarative definition instead of an operational definition. We describe the procedure used to install the packages and some related issues concerning software delivery; software configuration; and the actual software installation for workstations components and user components. Partial details of our implementation of System V packaging is included in the Appendix.

1 Introduction

To make the installation of software in the desktop computing environment of UBS, we have implemented a state driven software installation mechanism. We are using Microsoft NT workstation as the standard desktop with Netware servers or NT servers as the associated file and print (f&p) servers or as home servers for user data. The tools described have been productive use since the late 1997.

We have selected to "lock" down the desktop and the machine for the normal users. And having a standard set of applications which are "packaged" using our own packaging format similar to the "System V" packaging [SysV]. This format is used for delivery, configuration and installation of the applications (please see the appendix for more details). The packages are delivered via CD or SMS [SMS1.2] to servers on the local LAN in the branch office. Then is the software installed on the f&p servers and/or workstations (clients).

To ensure that all workstations have a standard set of applications installed, we store the configuration of each workstation in a "centralized database" [1] (this is not an inventory database). The configuration data defines pro workstation which packages should be installed, the installation mode, and the installation order between the packages. The users of the workstation is restricted to the set of packages listed in the configuration database for the workstation.

At well defined times, a NT service running on all NT workstations and NT servers wakes up or is awakened. The service compares the current state of the machine (which packages are actually installed) against the state defined the configuration data. The service then performs the necessary actions to reach the state as defined in the configuration data.

When the users login into "their" workstations, we compare the user components already installed into the user's profile (including home drive) against the set of packages installed on the workstation itself. We compute the difference between the state of the workstation and user's profile, and then perform the necessary actions to update the user's profile to the state of workstation[2]. We name this *flex seating* since the users can flex or change between the different workstations which are similarly configured, having access to all the standard application via their profile[3].

*The work described was done while the author was at UBS AG, Private and Corporate Consumer Division, between 1996 and 1998.

[1] The database is stored securely on a server, either as flat files, ini files, or in a RDBMS.

[2] In reality, it is more complicated, but the scope of the paper does not allow us to elaborate on the software authorization mechanism.

[3] To allow full roaming between different branch offices or different resource domains, we need to add support for on demand package installation.

For the NT servers, the same process applies as for the NT workstations, but with a single extension - the NT servers must also handle packages which are exported to the clients, e.g. dictionaries for Microsoft Office. Compare this against the packages which are installed for the NT server itself, such as SNMP, backup, virus scanner etc.

As seen from the above description, we have a declarative definition of what should be installed on each machine and by extension which application users have access to. By have the state of the machine defined in a configuration database, it is very easy to recreate the state of the machine after a crash[4]. Compare this with more "normal" operational definition where the state of the machine is defined by the set of application installed on the machine, possible together stored a central inventory of the machines.

2 Computing Environment

We support user authentication against a NT domain with home directory and profiles stored on a NT server, or against Novell's NDS where the user profile and home directory is stored on a local Netware 4.x server. For the typical environment in a branch office (the listed server are logical servers, often there will be a single physical server), we have:

- *Clients*, the workstation or desktop machines, which are all running Microsoft NT 4.0. All client have an associated shared application server.
- *Shared Application Server* which is where common components like dictionaries, templates, and seldom used applications are stored. This is often the print server for the associated clients. A few standard SMB shares are available for the client.
- *Home Server* is where the user home directory is mapped and stored together with the NT 4 user profile.
- *Package Server* stores the packages to be installed. Exports a SMB share with the *package spool area*. For a SMS distribution server we also have the standard SMS package share. For NT servers, we distribute package via SMS and for Netware server we distribute packages via monthly CDs. Or via HTTP or FTP in emergency cases.
- *Configuration Data Server* stores the "configuration database" as `ini` style file for the machines.

[4] We use the same mechanism for the initial setup of the machine by providing a small set of bootstrap tools in the OEM directory of the Microsoft unattended setup, and then invoking our installation process.

3 Implementation

The software installation (including both removal old versions and additions of new versions of software) can logically be split into the installation of:

- F&P servers components is the shared context of packages. These are made available for the workstation from its associated f&p server via SMB shares. For NT servers, the software installation is done by the "EUP Installer Service" (`eupsrv`).
- Workstation components includes the workstation context of a package for a *shared* mode installation, but can also include the shared context and the workstation context for the *local* mode installation. As for the NT server, the software is installed by the "EUP Installer Service" (`eupsrv`).
- User components are only the user contexts of a package, installed by by the `euplogon` program at user login time.

We treat the installation of server components and workstation components as a single case. From the actual calculations the server case is an extension of the workstation case to handle the shared context exported to the clients via the SMB shares.

For the installation of the server and workstation components, we need the configuration data describing the wanted or target state of the machine. In the configuration data for the machines are stored: the list of packages to be installed, their installation mode (local, shared, or exported), the location of the package spool, and control parameters for `eupsrv`. The installation order of the packages is the order of the packages in the configuration data.

3.1 Machine Components

The first step for the **eupsrv** is to determine if any users are logged in - we user several methods: a GINA [GINA], enumeration of open desktops, and a flag set by the **euplogon** at user login. If an user is present, we either present a warning dialog asking the user to logout as soon as possible, or we will retry within a pre configured time limit (defaults to 4 hours). Once the **eupsrv** can detected no users, it optionally login into the associated shared application server using the credentials provided in the registry configuration. This is to get access to the configuration data, to the shared application server, and to the package server.

The initial step is to determine the set of packages already installed on the machine, state (installed or removed etc.), mode (local, shared or exported), the installation order, and who installed them and when. By scanning the pkginfo directory, for a NT workstation and a NT server: `%SystemRoot%\Config\PkgInfo\`. And for the NT server the list of exported packages: `\\server\PkgInfo\`.

For each package we read the installed package's **pkginfo.ini** and **pkgvars.ini** files to retrieve the installation time, the shared application server, installation status (successful installed, partially installed, or removed):

```
[Status]
PSTAMP=MARTIN980226113655
UserId=SYSTEM
Status=Successfully Installed
Date=1999.11.11:23:05:00
```

The shared application server associated with each package is retrieved by reading the value from the **pkgvars.ini** file. The installation mode is *shared* if no shared context have been locally installed. If a shared context and workstation context have been locally installed, the package have been *locally* installed.

Thus, we have determined the set of current packages installed on the machine:

$$CurrentState = set\ of\ packages\ already\ installed \quad (1)$$

We connect to the configuration database to retrieve the target state, which is an ordered list of packages with their installation mode:

$$TargetState = set\ of\ packages\ in\ the\ configuration \quad (2)$$

Having *CurrentState* and *TargetState*, we compute the set of package necessary to remove from the machine, by taking all packages not present in the *TargetState* but currently found on the machine, *CurrentState*:

$$Pkgs2Remove = CurrentState \setminus TargetState \quad (3)$$

By default, we filter out package which have not been installed by the service from the *Pkgs2Remove* set, since removing the Emacs package which a developer have installed for his own use is not acceptable.

The next step is to compute the set of packages which we must install on the machine to reach *TargetState*, which is the set of all package present in the *TargetState* but not in the *CurrentState*:

$$Pkgs2Install = TargetState \setminus CurrentState \quad (4)$$

Notice that two entries when comparing are only considered equal if the following conditions are met:

1. Same package name (**ubsperl_5_un_1.4**),
2. Same installation mode (shared, local or exported),
3. Same shared application server for *shared* installation mode, and
4. Already installed package is installed successfully.

Once we have computed the *Pkgs2Install* and *Pkgs2Remove* sets, we order the *Pkgs2Remove* in the reverse installation time to avoid any problems with install time or run-time dependencies. The *Pkgs2Install* set should will the same order as specified in the configuration data. If both *Pkgs2Remove* and *Pkgs2Install* sets are empty we are finished.

Before we start the actual package addition or package removal, we verify for all package in the *Pkgs2Install* set that the package is actually present on the package server. We also verify that for shared mode installation that the correct version of the shared context of the package is installed on the the shared application server. If not both of these conditions are fulfilled, we will ignore the package during the actual installation.

We start traversing the *Pkgs2Remove* set and remove the already installed packages by calling `pkgrm`. For the shared mode, we remove the workstation context (which is the only context installed). And for the local mode, we first remove the workstation context followed by the shared context.

The following step is to process the *Pkgs2Install* set and to install the new packages by invoking the `pkgadd` for each package. The major parameters are: package name, shared application server, and path to package spool area. For locally installed packages, we first install the shared context and then the workstation context.

The `eupsrv` invokes *pre-* and *post-* scripts stored on the machine in a secure location and stored on the shared application server before the enumeration of the installed package as well as after adding the last package. This enable packagers or the local supervisor to modify the state of the machines and have to influence what are installed.

Last, for a package which requires an immediate reboot, as specified by the restart flag in the `pkginfo.ini`, the `eupsrv` will force a reboot of the machine once the package have been removed (or added). For packages which requires a reboot before becoming operational, the `eupsrv` will reboot after the last action (install or remove) has been done. The `eupsrv` will continue operations after the reboot, either being restarted by the SMS PCM (Package Command Manager) or by itself.

3.2 User Components

When the user login to a NT workstation, the `euplogon` program is invoked - we have replaced the standard `UserInit` registry value. The `euplogon` program compares the list of packages already installed on the machine (by scanning the machine's pkginfo directory) against the list of user contexts already installed to the user profile and home drive (by scanning the pkginfo directory stored on the user home drive and checking the cached values in the registry).

Using roughly the same computation as described in the previous section, the program determine the set of user contexts to remove and the set of user contexts to add to ensure that the user have the same set of packages as already installed on the machine.

One minor change is that we only remove a user context if the current machine where the user is logged in to is the same machine where we originally installed the user context. We avoid removing a package when an user temporarily login into another workstation than the her normal workstation. Instead, we hide any shortcut in the user program menus by setting the `HIDDEN` bit on the shortcut.

4 SMS Integration

Since one year, we support software distribution via SMS and also initiate software installation via SMS for the "pure" NT environment. This integration caused a number of changes or improvement in the `eupsrv`.

4.1 Push versus Poll

When we control software installation using SMS, we would like to initiate the software installation via a standard SMS job. This changes the operational mode from "poll" (check if configuration data have changes at regular intervals) to "push" (start now and verify if the state of the machine matches the state as defined the configuration data).

The standard operational mode of the `eupsrv` in the initial release was to poll the configuration data (a `ini` file) every fourth hours when a user was not logged in. To avoid re reading the file when it not have changed, we cache the last modification date and size in the `eupsrv` and compare those attribute to determine if the file have changed.

We have the `PCM` installed on all NT workstations, NT servers, and SMS distribution servers as a service. We extended the `eupsrv` to be started from

the PCM and no longer polling the configuration data at regular interval. When called from a SMS job, the **eupsrv** service is started from the PCM. The command line executable blocks until the **eupsrv** service returns with a status code before it exits (and thus blocks the PCM and the current SMS job).

The **eupsrv** was extend with three command line option to support:

- **eupsrv -run** which verify the state of the machine against the state as defined the configuration data. This command is invoked as part of a SMS job to force an update of a machine.
- **eupsrv -pkgadd** *package-id,...* to enable the installation of one or more package from a SMS job, *if* the packages are included in the configuration data for the machine. The installation mode is read from the configuration data.
- **eupsrv -pkgrm** *SMS-ID,...* to enable the removal of one or more packages from the target machine. No check is done for run time dependencies, so this can break an existing installation.

Some complication arise in the handling of packages which requires a reboot or who should restart the **eupsrv** after the reboot. The simple solution was to stop the SMS PCM service when a reboot was required, and having **eupsrv** forcing a reboot of the machine. After the reboot the PCM detects that the SMS job was not finished, and retries SMS job including the **eupsrv** command.

4.2 SMS Software Delivery

One major problem with SMS 1.2 is that the actual software distribution is not atomic to the distribution servers. By atomic delivery, we mean that either is the package on the SMS share to 100% or to 0%, not partially present.

To all SMS distribution jobs, we added a small batch script which created a flag file, in the root of the package directory once the package was delivered to the distribution server. The second step was to extend the **eupsrv** to verify if the package was fully delivered to the share by checking for the flag file.

4.3 Shared Contexts Coordination

For packages which are installed in shared mode on the workstations, we must ensure that the shared context is present on the associated shared application server and also that the correct version of the package is present. This was added as part of the SMS extensions to ensure that SMS jobs sent directly to the workstations did not try install the workstation context of the package before the shared context was present on the associated f&p server.

5 The Good, the Bad and the Ugly

We have learned a few lessons during the last years during the development of the installation mechanism, especially issues which crept up during the integration with SMS. What follows is a partial lists of points.

We have discovered bugs in both Microsoft Networking code and the Netware client for NT. The network and security issues to get the **eupsrv** working correctly have been causing headaches.

The "Poll" mode sounds good and looks good, but the local supervisor needs more control over when an software installation is started, which workstations should be done, and a mean to reduce the load on the network and/or the server. We solved this by adding a common configuration file on the local Netware server, but ...

Under Netware 4.x, we need to authenticate against NDS before we can read files on the server, e.g. the package or configuration files. In the case of the configuration file above, even though we had limited the number of parallel updates to, say 5, we overloaded the NDS authentication server when all the workstations tried to read the configuration file. For the Netware environment, a possible solution would to use a TCP server for common configuration data, or start the **eupsrv** via Netware Workstation Manager.

The current general policy for the installation of a new version of a package is first to completely remove the old version and then install the new version. But most of the new versions of packages are incremental improvement or small changes to the components. The current approach causes a

lot of extra network traffic by coping data from the package server which to a large extend was already present on the target machine. Either overwrite functionality or a delta package mechanism to reduce the network traffic as well as the installation time should be considered. The current "distributed" database of installed components, `pkgstat.dat`, in each package's pkginfo directory pro context, should be centralized to easy the implementation of overwrite packages.

Centralize the configuration data in a central repository (database) and distribute it using LDAP [LDAP] or similar. By extending the initial design with new configuration files on the shared application server, we added more and more configuration data in several location instead of going for a unified interface. With the new release using the *EUP Values* we have initiated the centralization and collection of the configuration data into a single location and single interface.

During the last year, we have put a lot of work into the creation of standard and guidelines for how to create "clean" packages, package creation tools, and package verification tools (against a central package database). This is absolutely essential to improve the quality of released packages and get a stable desktop platform.

For the future, with Office 2000 [Office2000] as well as NT 2000, Microsoft Software Installer [MSI, MSIT] (MSI) is looming on the horizon together with new release of InstallShield [InstallS] as well as SMS 2.0. We have started working on how we can replace part of our in-house developed components and integrate them with MSI.

Notice that the ideas expressed with a state driven installation can be realized using any installation format as long as the state is saved on the target system and in the user registry and/or home drive. There must also exist one-to-one mapping between a package version and the stored state to enable us to uniquely determine which version was installed.

6 Acknowledgments

The work described is the work of a lot of people over the last four to five years at UBS, and an incomplete list: Andre Aeppli, Martin Hufschmid, Diedrich Klarmann, Peter Kurz, Nick Riordan, Martin Schaible, Jeffrey Tolmie. And especially thanks to those I have forgotten.

References

[SMS1.2] R. Anderson, J. Farhat et al, *Microsoft SMS 1.2 Administrator's Survival Guide*, Sams Publishing, (1997).

[InstallS] *InstallShield for Windows Installer: Overview, White Paper*, Version 1.0, December (1998).

[LDAP] T. Howes and M. Smith, *LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol*, Macmillan Technical Pub., (1997).

[MSI] M. Kelly, *Gain Control of Application Setup with the New Windows Installer*, Microsoft Systems Journal Sept (1998) p. 15-27.

[MSIT] *Microsoft's Software Installation Technology. Part 1: Client-side Installation Service*, Directions on Microsoft (Research), March (1998).

[Office2000] *Microsoft Office 2000 Deployment and Maintenance, White Paper*, `http://www.microsoft.com/office`

[SysV] *System V Packaging Manual*, `http://docs.sun.com`

[Win5] *WinInstall version 5.1*

[GINA] *Winlogon User Interface*, Microsoft Win32 Software Development Kit for Microsoft Windows.

A A System V Style Packaging System for Windows NT

The installation mechanism described is build on top of a packaging system for Windows NT which have been implemented in house since the middle 90's.

The application format is an implementation of a "System V" [SysV] like packaging system as under Solaris/SunOS, with improvements to enable a

smoother integration with both the Windows NT and the Netware computing environment. We used concepts (`prototype` file, `pkgmap` file, etc.) from the System V packaging and also integrated features (variables and variable expansion in the output files) from the WinInstall [Win5, InstallS] product. We have kept the name of the standard packaging tools: `pkgmk` for package creation, `pkgadd` for package installation, `pkgchk` for package verification, and `pkgrm` for package removal.

We will try to give an overview of our packaging system as used in the environment with Windows NT clients connected to f&p servers. The target is to describe enough of the packaging system to make the installation mechanisms clear, without going into the more esoteric details of the actual implementation.

A.1 Background

Why packaging? Why not simply use WinInstall [Win5], InstallShield [InstallS], or the format as provided by the Microsoft SMS Installer [SMS1.2]?

When the Windows NT 4.0 project started in 90's, the existing technology was not good enough, and did not support both Netware servers as well as NT servers as target for application installation or parts thereof. We must support application installation to both server platforms and we had already a working packaging system for Netware servers. We also have experience with the System V packaging tools which we have extended extensively.

As installation targets, we must support NT workstations, NT servers, and Netware servers with a single packaging format for all platforms. We have a mixed environment where NT workstations can have a f&p server which is a NT server or a Netware server. And for the applications installed on the workstations, there must be no difference if the associated f&p server is a NT server or a Netware server.

Further, we must also support different languages on the server (English, German, French and Italian); flexible directory structure (not all application are installed into `C:\Program Files`; co-existing of 16-bit and 32-bit applications; configuration of application (e.g. where is the SQL server); and clean removal of applications.

A.2 Shared, Workstation and User Contexts

We generally talk about three different parts of a package: the shared context, the workstation context, and the user context. Before we go into details about the different contexts, we need to mention that a package can be installed in three modes:

Local mode is when everything in the package is installed on the target machine.

Shared mode is when part of the package is installed on the f&p server associated with a workstation. The typically installed components are dictionaries, help files, clip arts, or shared database files. In this case a set of workstations "share" the common items on the server.

Exported mode is only valid for a NT server (a shared application server) where a package is "exported" to a set of a workstations. The shared context of the package is installed on the server, but read/referenced by the clients.

It is important to notice that the package itself can be created (and should) in such a way to support both local and shared installation modes. Often the installation mode is selected at *installation time* and not at package creation time.

The *shared context* of a package are those files which are part of the shared installation. This can only be files and no registry entries, since the files are made available to the client via a SMB share[5] on the associated f&p server.

The *workstation context* of a package are file components and registry components which are installed on the client itself. The registry changes are made to the machine hive (HKLM[6]), often adding configuration information for the application or adding definitions for a DLL. The files are typically installed on the machine either in the `%SystemRoot%`, `%System32%`, or under `%SystemDrive%` directory. Typically a shared DLL (mfc42.dll) goes into %System32% and netscape.exe into `%SystemDrive%\ie_appl\ie_4\netscape`

[5] Typically `ie_appl`

[6] Hive Key Local Machine

The *user context* of a package contains the registry changes to the user hive (HKCU[7]), possible configuration files into the user home drive, or changes to the user profile. An typical example is to add a shortcut to the program menu, pointing to the application installed in the workstation context of the same package.

A.3 Packaging Classes

As under System V packaging, we provides a set of standard classes to perform the usual manipulations needed to install an application. All but one class is external, that is implemented as an executable outside of `pkgadd`:

none used for directory creation, file copying and also to copy the `install.txt` and the `pkginfo.ini` into pkginfo directory (internally implemented class).

registry class to do changes to the registry (`REG_SZ`, `REG_DWORD`, `REG_MULTI_SZ` etc.).

iniclass for changes to `.ini` files.

execbat to invoke `CMD` script during package add or package removal.

pkgenv to change the machine or user environment.

pkgpath to add or remove components to the machine (system) path or user path.

shortcut to create shortcut to an executable in the user's start menu or in the default/all menu for the machine.

pkgassoc to add an association between a file type (`.htm`) and an executable (`netscape.exe`).

pkghosts to add an hostname entry to `etc\hosts`

pkgserv to add a service definition to the `etc\service`

regocx to register a "self registration" DLL.

resolve to replace packaging variables in the input file with variable values (variable expansion).

[7] Hive Key Current User

In System V packaging, we have the concept of classes which are used to determine which part of a package is going to be installed. We do not have classes to do dynamical installation configuration, instead it is possible to place conditionals in the `prototyp.txt` as for the C pre processor (`#if... #else... #endif`). The conditional allow the package creator to query different package variables to determine which files to install. A typically examples is to install the correct sound card drivers dependent on the type of the machine.

A.4 Packaging Variables

In System V packaging, the `pkginfo` file is used together with the `request` script to gather the input necessary for the correct installation of the package. The environment created by the output from the `request` script can then be used during the `preinstall` and `postinstall` scripts.

In our implementation, we do not support interactive prompting (à la `request` script) for the install time configuration. All information must either be present on the machine at installation time, it must be possible retrieve from a configuration file, or it is possible to compute the configuration information.

We have the `pkginfo.ini` file. The following is an example from the standard UBS Perl package:

```
[PKGINFO]
PackageName=UBSPerl
PackageVersion=1.4
Description=Perl-Win32
ProgramName=perl
ProgramVersion=5
ProgramVendor=GNU
VendorVersion=5.004p2
ProgramLanguage=UN
PackageCategory=1
PackageType=0
PackageArchitecture=W32
InstallType=FULL
TargetArchitecture=LAN,STD
DiskSpace=6729603
DiskUser=0
DiskShared=0
DiskWorkstation=6729603
Restart=No

[R-Dependencies]
RT_MFC>=4,UN

[I-Dependencies]
RT_SYSTEM=4,IE

[Status]
PSTAMP=MARTIN970814135523
```

The variables listed in the section 'PKGINFO' are available during the whole packaging installation

process. Further, for each machine, we have a packaging variable configuration file, **pkgvars.cfg**, which contains the standard mapping between the defined official variable names and the locations. This file also define the standard menu name and structure, the standard protection code for shared, workstation and user files. As an example for a Netware server with international English release:

```
System32        = "%SystemRoot%\system32"
UserRoot        = "%UserDrive%"
PackageData     = "%UserRoot%\%PackageName%"
AllUsers        = "%SystemRoot%\Profiles\All Users"
UserDrive       = "H:"
PackageDir      = "%ApplRoot%\%PackageID%"
UserMenu        = "%UserProfile%\Start Menu"
StartupMenu     = "%ProgramMenu%\Startup"
ReleaseDir      = "IE_APPL"
ProgramMenu     = "%UserMenu%\Programs"
DeveloperMenu   = "%ProgramMenu%\Development Tools"
DocumentsMenu   = "%DeveloperMenu%\Documents"
OfficeMenu      = "%ProgramMenu%\Office Applications"
PersonalMenu    = "%ProgramMenu%\Personal"
ApplDrive       = "I:"
ApplRoot        = "\\%server%\SYS2\%ReleaseDir%"
```

By having different configuration files for NT servers and Netware servers, we hide the differences in the directory structure. We also define in the guidelines which variables can be used and in which context, e.g. in the shared context your are not allowed to install a DLL into **%System32%** directory. You can only install a DLL into this directory in workstation context.

The variables can be used in the **prototyp.txt** and in all the files which are input to the external classes. By using variables in the file input to **registry** class, we will have the correct path to the executable:

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\\Perl]
'PERLLIB'='%SystemDrive%\\UES_Tools\\%ProgramName%\\lib;'

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.pl]
@='Perl'

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Perl\Shell\Open\Command]
@='perl.exe %1 %*'
```

So far, the packaging variables described only have allowed configuration based on the static settings stored on the target machines, using the configuration files and the installation target. We have extended the packaging system to allow installation time querying of variables values, so called *EUP Values*. The actual value of the variable can be a single string value or multiple values. All variables which begin with a standard prefix are queried at installation time by **pkgadd** via an well defined interface exported by a DLL[8].

[8] By exchanging DLLs, we can have different data sources and transport mechanisms.

A.5 pkgmk

As under System V, the main input to the **pkgmk** is the **prototyp.txt** file which describes the components as well as into which context the different components are to be installed:

```
[Shared]
d none "%PackageDir%" ? ? ?

[Workstation]
!search ".\install"
e execbat "preTask.cmd" ? ? ?
!search ".\system32"
e registry "wks.reg" ? ? ?
f none "%System32%\nsrt2432.acm" RO ? ?
c #if %EUP_MachineType%=SERVER
  f none "%System32%\rt32cmp.dll" %WksFileAttr% PD ?
c #else
  f none "%System32%\rt32dcmp.dll" %WksFileAttr% P ?
c #endif
e execbat "postTask.cmd" ? ? ?

[User]
!search ".\install"
e registry "user.reg" ? ?  D
e shortcut "shortcut.sct" ?  ?  ?
```

The **pkgmk** generates a spooled package in a spool area with a standard directory layout. Included is the file **pkgmap.dat** describing the contents of the package which corresponds to the System V **pkgmap**.

A.6 pkgadd

For the installation of a package, or rather a context of package, we use **pkgadd**, specify the full package name, the package spool area, the target machine, and the selected context. Before we install the workstation context we must have a shared context installed, either locally on workstation or the associated the f&p server. This also applies for the user context, which cannot be installed until the workstation context have been installed.

Under System V packaging, we have the **preinstall** and **postinstall** standard scripts which we can use to prepare and/or for the cleanup of the package installation. This will have to be implemented via **BATCH** scripts located in the **prototyp.txt** in the first or last position within in each contexts. For example, see the **postTask.cmd** and **preTask.cmd** in the example **prototyp.txt**.

Under System V, the installation of a package modifies the **/var/sadm/install/contents** to add the

newly installed components and their protections together with the owner of the components. Also, a reference count of the packages which have installed the same components are kept in the **contents** file.

Instead of having a central database (the **contents** file), we selected to have a distributed state pro package and pro context in the local file system on the target machine and in the user home drive/ registry (part of the user profile). Notice that the same format and contents is used for all contexts and all targets. When the package is installed, **pkgadd** creates a "package information" (pkginfo) directory in the proper location on the target (machine and context) using the package name provided as argument to **pkgadd**. For a shared installed package, we will have a pkginfo directory for the shared context on the server; a pkginfo directory for the workstation context on the NT workstation; and a pkginfo directory in the user home drive and registry. Typically, the different locations for the example package **ubsperl_5_un_1.4**:

- `\\zhsv13664\PkgInfo\ubsperl_5_un_1.4\` for the shared context installation on a NT server.
- `\\zhbdvf02\Sys2\PkgInfo\ubsperl_5_un_1.4\` for the shared context installation on a Netware server.
- `%SystemRoot%\Config\PkgInfo\ubsperl_5_un_1.4\Shared\` for the shared context installed on a workstation (local install).
- `%SystemRoot%\Config\PkgInfo\ubsperl_5_un_1.4\` for the workstation context[9].
- `%HOMEDRIVE%\Sys32\PkgInfo\ubsperl_5_un_1.4\` for user context of the package[10].

In the pkginfo directory, we have the **pkginfo.ini** file, which is the same as provided in the package spool, except the information about the installation have been appended (when was the installation done, status, and who did the installation).

Further, we find the **pkgvars.ini** file, which contains the actual packaging variables, and their values, used for the installation of the package. We also have the **pkgstat.dat** file which contains the list of components installed in actual installation order.

All files which are used as input to any of the classes are also kept (e.g. input to *registry* class, **path** class etc.). Notice that the input files in the original package is processed and all the package variables are replaced with their values before the files are feed to the classes. By keeping the input file to the classes, we will not need the package spool during the package removal. Second, we ensure that the same values for the packaging variables are used for the package removal as were used for package installation.

A.7 pkgrm

For the removal of a package, we have **pkgrm** and all the necessary information for removal is present on the machine in the pkginfo directory. The package contexts for a single package should be removed in the reverse installation order, e.g. for a *local* installation the workstation context should be removed before the shared context. Notice that we should not remove the shared context installed on a server until all associated workstation have removed their workstation context. User contexts, being the last installed context for a package, can be removed at any time.

The **preremove** and **postremove** supported by System V have to be replaced with the **PreTask.cmd** etc. Notice that the installed components within the installed package are removed in reverse installation order using the **pkgmap.dat** file stored in the pkginfo directory.

[9] The location is stored in the registry on the machine and set when the machine configured. No hard coded paths are stored in tools themselves

[10] The exact path is stored in user registry and is initialized the first time **pkgadd** is called.

IP.com PriorArtDatabase

Technical Disclosure

Non-default Software Installation Technique

Author(s): **IBM**

IP.com number: **IPCOM000020584D**
Original Publication Date: **November 28, 2003**
IP.com Electronic Publication: **November 28, 2003**

IP.com, Inc. is the world's leader in defensive publications. The largest and most innovative companies publish their technical disclosures into the IP.com Prior Art Database. Disclosures can be published in any language, and they are searchable in those languages online. Unique identifiers indicate documents containing chemical structures as well as publications open for comment in the IP Discussion Forum. Disclosures are published every day online and also appear in the printed IP.com Journal. The IP.com Prior Art Database is freely available to search by patent examiners throughout the world.

Client may copy any content obtained through the site for Client's individual, noncommercial internal use only. Client agrees not to otherwise copy, change, upload, transmit, sell, publish, commercially exploit, modify, create derivative works or distribute any content available through the site.

Note: This is a pdf rendering of the actual disclosure. To access the notarized disclosure package containing an exact copy of the publication in its original format as well as any attached files, please download the full document from the IP.com Prior Art Database at: http://www.ip.com/pubview/IPCOM000020584D

www.ip.com

Copyright © IP.com, Inc. All rights reserved.

Non-default Software Installation Technique

Different installation and package managers have different capabilities and limitations. For instance, Installp package manager on AIX does not support relocation of product filesets nor does it support installing multiple instances of a given fileset on a given system, a feature that is important in allowing customers to perform compatibility tests before integrating new versions of a product into a production environment.

Obvious solutions would be to choose an installation and package manager that provides all the necessary features required by the target customer. However, to comply with corporate guidelines and 'cultural' conventions on a given platform, at times one does not have a choice in package managers. For instance, if it is a product policy to provide installation packages in the standard format supported by the operating system (for example, RPM on Linux, InstallShield on Windows, Installp on AIX, etc), deviating from the standard package manager is not an option.

A solution is required to allow a product to distribute its packages in the standard format that is supported by the standard package manager, and while still have a means to provide alternative means of installation using the same product packages.

The non-default software installation (NDI) technique provides the user the flexibility to use an alternative means of installing product packages. By bypassing the default package manager, the developer is able to work around any limitations imposed by the package manager. For instance, using the NDI installation facility, one is able to bypass AIX Installp's restrictions on relocation and installing multiple instances of product packages by not logging entries into the AIX ODM (Object Data Management) database.

Essentially NDI works by manually extracting files from the product packages and stores them on the system as appropriate for the product. Wrapper scripts are created around the product executables to export the necessary environment to mimic a 'standard' installation.

The advantage of using NDI is that the product developer is free to provide the necessary installation functionality regardless of the format of the product packages as required by the standard installation.

The NDI tool takes a list of the product packages and manually extracts each file from the package's file archive to the user specified directory to create a product tree that has a user defined 'root' (which over comes any restrictions on relocation of product packages).

AIX Installp Fileset Example:

```
# determine list of package files:
% cd $working_directory
% restore -qvf $package ./usr/lpp/$package/liblpp.a
% ar -x ./usr/lpp/$package/liblpp.a
% parse $package.inventory
# manually extract package files:
cd $root
for $file in @package_inventory
do
   restore -qvf $package -x $file
done
```

In order to facilitate the proper execution and functionality in the relocated directory, product invocation commands are "wrapped" by shell scripts. The role of these wrapper scripts is to set up the appropriate environment variables and usage of the correct configuration file which reflects the relocated product tree before invoking the real executable.

Compiler 'cc' Invocation Command Wrapper Script Example:

```
#! /usr/bin/ksh
export PATH=$root/opt/product/bin:${PATH:-/usr/bin:/bin:/usr/sbin}
export NLSPATH=$root/usr/lib/nls/msg/%L/%N:${NLSPATH:-/usr/lib/nls/msg/%L/%N}
export LIBPATH=$root/opt/product/lib:${LIBPATH:-/usr/lib:/lib}
exec $root/opt/product/bin/.orig/cc -F$root/etc/product.cfg "$@"
```

The wrapper script would include an argument to pick up a configuration file which has been tailored to reflect the relocated product tree (can be created as part of the NDI process). The combination of the wrapper scripts and configuration file ensure that the correct executables, libraries and header files are used in the event that multiple instances of the product are installed on the system.

IP.com
PriorArtDatabase
Technical Disclosure

Method to install/update USB drivers

Author(s): **Disclosed Anonymously**
IP.com number: **IPCOM000007916D**
IP.com Electronic Publication: **May 3, 2002**

IP.com, Inc. is the world's leader in defensive publications. The largest and most innovative companies publish their technical disclosures into the IP.com Prior Art Database. Disclosures can be published in any language, and they are searchable in those languages online. Unique identifiers indicate documents containing chemical structures as well as publications open for comment in the IP Discussion Forum. Disclosures are published every day online and also appear in the printed IP.com Journal. The IP.com Prior Art Database is freely available to search by patent examiners throughout the world.

Client may copy any content obtained through the site for Client's individual, noncommercial internal use only. Client agrees not to otherwise copy, change, upload, transmit, sell, publish, commercially exploit, modify, create derivative works or distribute any content available through the site.

Note: This is a pdf rendering of the actual disclosure. To access the notarized disclosure package containing an exact copy of the publication in its original format as well as any attached files, please download the full document from the IP.com Prior Art Database at: http://www.ip.com/pubview/IPCOM000007916D

www.ip.com

Copyright © IP.com, Inc. All rights reserved.

Method to install/update USB drivers

Disclosed is a method to install/update USB drivers. Benefits include improved performance and reliability in the installation/update process, which will reflect in the proper functioning of the USB devices that use the installed drivers.

General description

The disclosed method provides a mechanism for programmatically installing/updating USB drivers on personal computer platforms. The methodology uses the following key elements:

- Initialization (.ini) file with specific driver information
- Set of driver files:
  - One or more information (.inf) files for the driver installation
  - Signature (.cat) file (required for logo certification)
  - Other driver related files (for example, .sys, .dll, and .ocx)
- Program scripts that perform the actual installation/update of the drivers by calling the appropriate platform API functions.

A key part of this methodology is the structure of the information provided in the initialization (.ini) file. The program uses this data and applies it to the installation/update of any USB device.

Advantages

This methodology provides a mechanism for the successful installation/update of any USB device, especially for products targeted to novice users. Without this methodology, the user would have to either:

- Manually install the device from the hardware wizard, locate the drivers, and run the risk of selecting either the wrong set of drivers for the specific platform (most drivers are platform specific), or possibly selecting the drivers from the wrong language (in the case of products that include drivers for multiple languages).
- Have a customized script specific to the device to be installed/updated.

This methodology offers the advantage that is generic, so any USB device can use the same program for the installation. Most installations use customized scripts, so there is duplicate work when changes need to occur, such as when a device uses multiple .inf files or when multiple Vendor IDs/ProductIDs (VIDPIDs) are referenced within the same .inf file.

By structuring its key data in the .ini file to provide the required drivers files, this methodology offers a simple and straightforward mechanism to install/update any driver without having to write a single line of code.

The disclosed method takes into account the possibility that different operating systems might have different formats for .inf files or driver files. The method also takes into account the fact that the same .inf file can reference multiple VIDPID numbers.

Detailed description

The disclosed method includes a program for installing/updating USB drivers on personal computer platforms (see Figure 1).

The Initialization data structure provides specific information about the device that needs to be installed (see Figure 2).

The SKU section contains the following values:

- DeviceID: Character field that identifies the device
- Path: Directory path where the driver files are initially copied on the user's hard drive by the installation
- VendorID: Device specific vendor identifier
- ProductID: Device specific product identifier
- DevRegistry: List of registry keys created when the device has been installed at least once and is used to detect update scenarios

The INF section contains information regarding the .inf files that are required for a complete installation of the drivers. Some devices require multiple .inf files. For instance, some digital cameras have up to three .inf files that are required for the camera driver installation. Different .inf files may be required for various operating systems. The INF section is a comma-delimited list with no limitation on the number of elements in the list. Examples of .inf files include 2Kinf and XPInf.

The FileN.inf section contains information related to each specific .inf file in the INF section.

- VidPid: VendorID/ProductID referenced in the .inf file

When multiple VIDPIDs exist, the list is comma delimited.

Driver files are comprised of three parts:

- .inf file(s): Information file that runs when a plug-and-play event is invoked, such as when the device is plugged into the USB port. They perform the actual installation of the driver.
- Signature file: File that verifies that the device has been qualified for use on the platform.
- Driver files: Set of files that are required for the device to work on a specific platform. These files can be provided as a list of loose files (.sys, .dll, or .ocx files) or as a cabinet (.cab) file that contains required files.

Installation programs are developed in an installation script. They process the device data provided in the initialization file and make the corresponding calls to the platform API to do the actual installation/update of the drivers.

Two types of API calls are required for the installation of the driver. One is used in the case of a clean install, meaning that the device has not been installed before. The other call is used when the device has been previously installed. The install script determines the appropriate API calls to install/update the device drivers correctly.

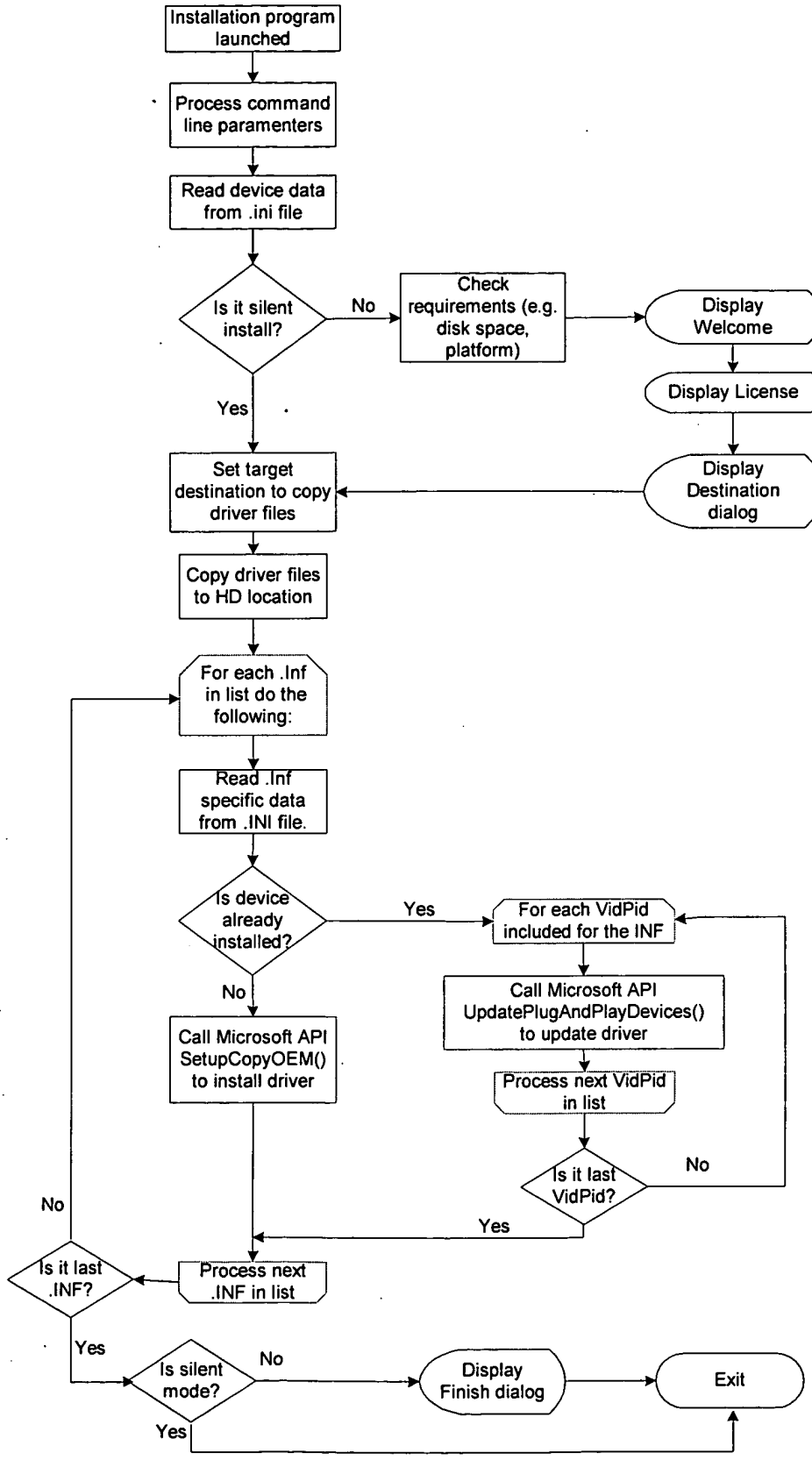
Installation program launched
Process command line paramenters
Read device data from .ini file
Is it silent install?
No
Check requirements (e.g. disk space, platform)
Display Welcome
Display License
Display Destination dialog
Yes
Set target destination to copy driver files
Copy driver files to HD location
For each .Inf in list do the following:
Read .Inf specific data from .INI file.
Is device already installed?
Yes
For each VidPid included for the INF
Call Microsoft API UpdatePlugAndPlayDevices() to update driver
Process next VidPid in list
Is it last VidPid?
No
Yes
No
Call Microsoft API SetupCopyOEM() to install driver
Is it last .INF?
Process next .INF in list
No
Yes
Is silent mode?
No
Display Finish dialog
Exit
Yes

Fig. 1

```
[SKU]
DeviceID =
Path =
VendorID=
ProductID=
DevRegistry=RegKey1, RegKey2


[Inf]
2KInf = File1.inf,File2.inf,… FileN.inf
XPInf = File1.inf,File2.inf,… FileN.inf


[File1.inf]
VidPid = VID_XX&PID_01,…,VID_XX&PID_0N

2KKeyFile= KeyFile1,…,KeyFileN
XPKeyFile= KeyFile1,…,KeyFileN
.       .
.
.
 [FileN.inf]
VidPid = VID_XX&PID_01, VID_XX&PID_02

2KKeyFile= KeyFile1,…,KeyFileN
XPKeyFile= KeyFile1,…,KeyFileN
```

Fig. 2

Disclosed anonymously

Mkpkg: A Software Packaging Tool

Carl Staelin
Strategic Planning and Communications
HPL-97-125 (R.1)
August, 1998

software packaging,
software distribution,
software publishing

Mkpkg is a tool that helps software publishers create installation packages. Given software that is ready for distribution, mkpkg helps the publisher develop a description of the software package, including manifests, dependencies, and post-install customizations. Mkpkg automates many of the painstaking tasks required of the publisher, such as determining the complete package manifest and dependencies of the executables on shared libraries. Using mkpkg, a publisher can generate software packages for complex software such as TeX with only three minutes effort.

Mkpkg has been implemented on HP-UX using Tcl/Tk and provides both a graphical and command line interface. It builds product-level packages for Software Distributor (SD-UX).

© Copyright Hewlett-Packard Company 1998

Mkpkg: *A software packaging tool*

Carl Staelin

January 14, 1997

Abstract

Mkpkg is a tool that helps software publishers create installation packages. Given software that is ready for distribution, mkpkg helps the publisher develop a description of the software package, including manifests, dependencies, and post-install customizations. Mkpkg automates many of the painstaking tasks required of the publisher, such as determining the complete package manifest and dependencies of the executables on shared libraries. Using mkpkg, a publisher can generate software packages for complex software such as TeX with only three minutes effort.

Mkpkg has been implemented on HP-UX using Tcl/Tk and provides both a graphical and command line interface. It builds product-level packages for Software Distributor (SD-UX).

1 Introduction

Most end-users do not build programs from source code, but install software using binary installation packages. Mkpkg helps software publishers develop those installation packages.

Mkpkg addresses a part of the software distribution channel that has been largely ignored. Most software distribution systems have focussed on defining the binary package format and the protocols for installing and de-installing software. Most software installation suites have made it very easy for end-users and system administrators to distribute and install software, but they have not addressed the problems of the software packager who is creating binary installation packages.

The software publishing process includes several actors and steps: the software developer, the software publisher, the distributor, (sometimes the system administrator), and the end-user. The software developer creates the software. The packager is responsible for configuring, compiling, and packaging the software to create the binary installation package that the distributor delivers to the end-user. In some environments system administrators install and manage the software for end-users.

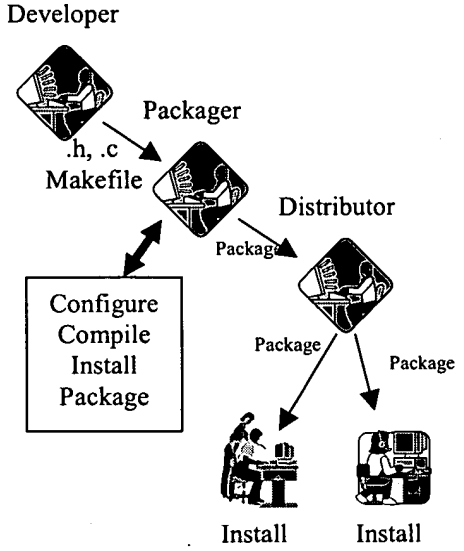

Figure 1

Mkpkg helps software packagers create installation packages. Typically, the packager starts with source code that needs to be compiled and installed on the packager's computer. The packager tries to create an installation package that re-creates the installation on each end-user's computer.

Developing a binary software installation package, that is, creating a package that can be installed easily on a computers and have it work properly, is an important and difficult task. Most vendors have developed tools that can accept the descriptions of a software package and create an installation package, but developing those package descriptions is difficult. Package descriptions typically contain the following elements:

Each software installation tool has its own idiosyncrasies and requirements, but they all share these common elements. Many software installation tools also provide other elements, such as system specifications that define which hardware/OS types or versions may install the software.

Element	Description
title	Software package name
description	A text description of the package and its capabilities
manifest	A list of all the files contained in the package
dependencies	A list of all the other packages required for this package to operate correctly
customization scripts	A set of scripts that are executed on the user's machine during installation or de-installation of the software

Table 1

1.1 Software Distributor

Software Distributor (SD-UX) is a suite of software installation programs that satisfy the POSIX draft 1387.2 specification. It is the software distribution mechanism for all Hewlett-Packard software for HP-UX and has versions that run on at least WindowsNT and Solaris.

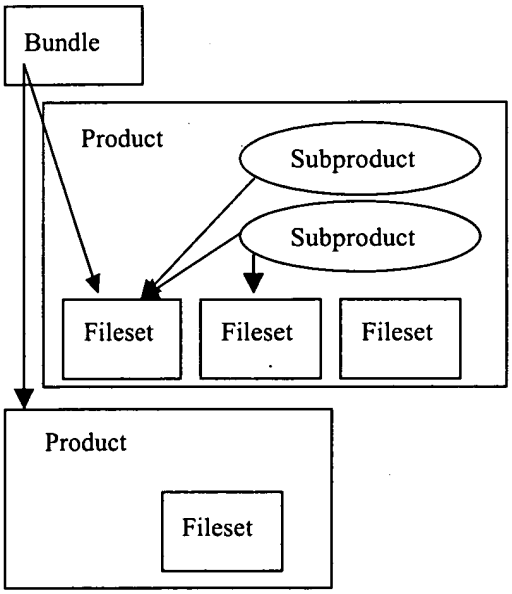

Figure 2

Software Distributor has four levels of software grouping: bundle, product, subproduct, and fileset. A bundle is a collection of products and/or filesets that may be installed as a unit. Bundles were designed to provide customers with one single installation unit for purchased software products, such as the ANSI/C compiler. Bundles may be used to provide a logical grouping by function, such as "web server".

The basic unit of software distribution is the product. A product may contain both subproducts and filesets. Subproducts contain filesets and are used to manage logical subsets of a single product.

Filesets are the atomic units of software distribution and contain a set of files and control scripts. SD-UX installs and configures individual filesets; filesets cannot be partially installed or configured.

Software Distributor has two levels of software distribution: bundle and product. The basic unit of distribution is the product. Software Distributor has several levels of software installation. The basic unit of installation is the fileset, but customers usually install software at the bundle or subproduct level.

Mkpkg creates product packages, while mkbdl creates bundles. Since all filesets and subproducts are created as part of a product, we do not provide a separate tool for creating them.

1.2 Installation tools

There are many methods for distributing binary installations. Each method has various strengths and weaknesses, but most commercial systems provide a similar level of basic operation. The largest UNIX vendors have each developed their own systems for distributing binary software: HP's HP-UX uses Software Distributor, Sun's Solaris uses *pkgadd*, Digital's OSF/1 uses *setld*, SGI's IRIX uses *inst*, and Linux uses RPM. Windows has two standards, InstallShield packages and self-extracting programs.

Each of the commercial systems offers basic services, such as installing and deinstalling packages atomically, tracking and managing inter-package dependencies, and executing scripts during software installation and deinstallation. Most of them also support a variety of more advanced features, such as versions, operating system and hardware dependencies, and interactive installation. For the most part these systems try to make it as easy as possible for system administrators to install software.

1.2.1 Tar

The simplest binary installation package is simply a tar file containing the software. Often such packages include a README file that includes installation instructions. For simple programs and packages, tar files are often sufficient. For more complex packages, much of the burden of correctly installing

and configuring the software falls on the end user because the installation process for tar files is completely manual.

Occasionally, software publishers will include installation scripts as part of the tar file and the installation script will automatically install and configure the software for the user. One of the few publishers who uses this approach is Netscape, who includes an "ns-install" script as part of their Netscape Communicator tar-file distribution.

1.2.2 RPM

The RedHat Package Manager (RPM) [2,3] was developed for the Linux environment and provides a very nice environment for installing and distributing software. Functionally, it is very similar to Software Distributor; it includes support for inter-package dependencies and control scripts that are executed during software installation.

Users may install software from a local depot or they may install from a remote server over the network. RPM is able to contain binaries for multiple platforms within a single package, and it can automatically install the correct binaries. Using RPM customers may determine which package installed a particular file, and what software is installed on the machine. Users may also uninstall packages.

RPM has some support for the packager, but it is missing some important features. RPM does not help the publisher develop the package manifest. The RPM documentation [3] states: "RPM has no way to know what binaries get installed as part of `make install`. There is *NO* way to do this. Some have suggested doing a `find` before and after the package install. With a multi-user system, this is unacceptable as other files may be created during a package building process that have nothing to do with the package itself."

1.3 Software configuration

Software configuration is one of the dirty little secrets of system administration. Software that is well configured works well in a broad variety of system configurations and causes few problems for system administrators. Poorly configured software can cause system administrators a great deal of aggravation.

Sometimes the difference between well-configured software and poorly configured software is a matter of tiny details, but there are a few general guidelines.

- Never compile paths into binaries.
- Separate executables, configuration files, and data or log files.
- Use human readable ASCII files for configuration information.
- Follow standard conventions for file and path names as much as possible.

System administrators frequently share file systems between systems, so executables and libraries will often be shared by many systems. However, administrators usually want each computer to have its own version of configuration files, but these may be on a read-only file system. In addition, data and log files are usually not shared between systems and usually must be mounted with read-write permissions. Software configurations must anticipate these kinds of configuration issues.

A fairly detailed set of configuration guidelines is published by the HP-UX Porting and Archive Centre [4].

2 Software packaging process

During software packaging, the publisher must prepare all the elements needed by the installation package. For many small packages, this is a very simple process, but for larger packages it can be quite difficult. Mkpkg provides five services during the packaging process:

- Creates the manifest, the list of files to be installed
- Determines the dependencies of this package on other packages
- Develops the install/de-install scripts
- Gathers all the components, as listed in the manifest
- Assembles and produces the completed installation package

2.1 Create manifest

The manifest is a list of all the files to be installed by the package. Mkpkg can automatically determine which files were installed by the package on the publisher's machine. For small packages it is easy to determine which files belong to a given package, but manual techniques often miss files and make mistakes. For larger packages, such as X11R6 or TeX, it is usually difficult to identify all the files

installed by the package and it is critical to include all the files that belong to the package.

2.2 Determine dependencies

Many packages require the presence of other software in order to operate correctly. For example, if **cvs** uses **rcs**, then **cvs** depends on **rcs**. Packages can depend on other packages for many reasons, but the two most common dependencies are caused by executing programs and linking with shared libraries from other packages. Publishers are usually aware of the dependencies caused by executing programs, but often overlook shared library dependencies.

2.3 Develop scripts

Typically, installation tools allow the publisher to add both scripts that are executed by the installation tool during software installation and scripts that can be executed during de-installation to undo actions and erase all trace of the software. Also, some software packages require customized scripts to handle special configuration during the installation process. For example, many database systems require a special userid to be added to the system.

Writing these scripts is very difficult, but many actions can be specified in a general fashion. In order to simplify the development of scripts, the publisher simply specifies the desired results of executing the script, and mkpkg generates all the scripts needed for the package.

2.4 Gather components

Once the package is specified, mkpkg gathers all the components, such as the customization scripts and installed files, and saves them in a temporary location. In the case for which multiple versions of a single package may be built (e.g. one version for statically linked binaries and another for dynamically linked binaries), the system may generate multiple copies of the system and save them in different locations.

If the user has added files to the package manifest, then the user may modify the package configuration after the components have been gathered.

2.5 Assemble package

The last step is to assemble the installation package from the package configuration, customization scripts, and saved installation. Mkpkg creates a Product Specification File (PSF) and all the automatically generated customization scripts and then uses swpackage to assemble and generate the completed package.

The PSF describes all the elements, options, and content of the installation package and is used by Software Distributor during package creation. Before mkpkg the PSF was nearly always generated manually.

3 Automation

Since many of the tasks associated with building binary installation packages are structured and are common across packages, it is possible to automate most tasks. Accurate automation has the benefit of increasing the uniformity of package configuration and operation across packages.

Mkpkg has automated or partially automated the following tasks:

- package manifest generation
- shared library dependency detection
- fileset and subproduct generation
- assigning files to filesets
- control script generation
- error checking

3.1 Package manifest generation

The first task faced by most package creators is creating a manifest or list of all the files installed as part of the package. It is critical that all files be included in the package, so it is important to reduce human error. For some packages, creating a manifest is a trivial task that can be accomplished easily by visual inspection of the software. However, packages often include dozens of files, and some packages include thousands of files. In these cases, it is very difficult to manually generate a complete and accurate manifest.

Mkpkg can automatically generate a package manifest that includes all files installed as part of the software and may include some files not belonging to the package.

3.2 Shared library dependency detection

Mkpkg automatically detects all shared library dependencies. It checks every file in the product to

discover which shared libraries are used by the product. It has a list of all shared libraries on the system and the name of the fileset that contains each library. Mkpkg then automatically marks as a co-dependency each fileset containing a shared library needed by an executable.

When I first developed mkpkg, the vast majority of bugs were caused by shared library dependencies that I had overlooked. Once I added this module to mkpkg the number of bug reports diminished dramatically.

3.3 Fileset and subproduct creation

Software Distributor allows a given product to contain filesets and subproducts (groups of filesets).

Hewlett-Packard has extensive standards for fileset and subproduct naming and semantics. For example, English-language manual pages should be contained in the *XXX*-**MAN** fileset, while foreign-language manual pages should have a fileset per language (e.g. *XXX*-**SPA-I-MAN** for Spanish with an ISO character set). Fortunately, it is possible to use simple regular expression patterns to recognize when particular filesets are needed. Similarly, there is an extensive set of standards for subproduct naming based on the filesets in a product (e.g. the subproduct ManualsByLanguage always includes all filesets with non-English manual pages).

Mkpkg has two ordered sets of rules for determining when to create filesets and subproducts. Each rule contains a regular expression, a threshold value, and a pattern. During fileset creation, the system iterates through the rules. It first creates a list of all files that match the regular expression. If the number of files is greater than the threshold value, then a fileset is created using the pattern (if necessary), and all the matching files are assigned to the fileset. The threshold value is used because some of the conventions are of the form "if there are enough XXX files, then put them in **–YYY** fileset". For example, "if there are enough manual pages, put them in a **–MAN** fileset".

3.4 Assigning files to filesets

The same rules that determine when to create filesets are used to assign files to filesets. This is particularly useful for large packages for which manual assignment of files to filesets would be tedious.

Mkpkg uses the same pattern matching to decide how to assign each file to a fileset. Each file is assigned to the first fileset whose pattern matches the file name. By default all files that do not match any fileset are assigned to the **–RUN** fileset.

3.5 Control script generation

One of the most difficult tasks is developing all the control scripts that customize the remote system. Fortunately, most control scripts execute a handful of common tasks, and in many cases it is possible to automatically detect the need for these tasks.

Control scripts may be used at both the product and the fileset levels. Mkpkg currently knows how to automate ten common tasks and allows the user to specify customization actions at either the product or the fileset level. The user specifies high-level actions that mkpkg maps into low-level script fragments for each of the ten possible control scripts. Mkpkg only generates control scripts when necessary; it doesn't generate empty control files.

3.6 Error checking

In general, it is very difficult to perform error checking for binary packages. However, there are a number of common errors that can be detected. Mkpkg flags as many errors as possible, but there is still room for "pilot error."

Each attribute of a product, subproduct, or fileset can be marked as "required". Before assembling the package, mkpkg can check that every required attribute has an associated value. For example, the attribute "description" is required and mkpkg will generate an error if this attribute has been left blank.

Mkpkg can also check that hard links do not cross fileset boundaries. In other words, if two files are joined by a hard link, then they must be in the same fileset. Optionally, mkpkg can ensure that symbolic links do not cross fileset boundaries.

4 Manifest generation

Mkpkg can automatically determine the product's manifest. In practice it is very accurate, but there are some occasional errors. There are two types of errors: excluding necessary files and including unrelated files. The most common problem is including unrelated files, and I have only had one package that did not include a necessary file.

Mkpkg creates the manifest using file timestamps to detect files that were installed by the install process included with the software source. Mkpkg creates a new file that it will use as a timestamp, then it builds

and installs the software (on the publisher's machine using the install process included with the software source). It then searches (part of) the file system for files with modification or creation times that are newer than the saved timestamp.

There are two types of errors that occur during manifest generation: missing files, and including extra files. In practice, mkpkg makes very few mistakes, but it cannot guarantee perfect accuracy, so the publisher must still verify the manifest's accuracy.

Since the manifest generation process uses file timestamps, it reliably detects all modified or installed files in the search region. There are two ways that mkpkg can miss files that should be included: directories missing from the search list, and files are that aren't installed. Sometimes, software installation tools are "too smart" and don't re-install files that have already been installed. In this case, some files will not be installed by the tool and will not appear on the manifest. I do not have a solution to this problem.

More commonly, extra files will appear in the manifest because the files have been modified independent of the installation process. Usually, these files are log files for system events or daemon processes. In general, the list of such files is fairly static for any given machine, so we can create a list of all such files, although Mkpkg automatically eliminates most of those files by automatically removing "spurious" files from the manifest. Mkpkg has a global list of "spurious" files that can be modified by publishers to match the active log files on their machines.

Mkpkg has a default list of directories to search for new software. Currently this list includes paths from *the Software Configuration Guide* [4]. Mkpkg will only search this part of the file system for newly installed files both to reduce the probability of independent user activity generating spurious file listings and to minimize the time required to search the file system for new files.

5 Control script generation

One of the most difficult tasks when developing installation packages is developing the customization scripts. These scripts are not interactive, may only rely on a restricted subset of system functionality and have to work correctly every time or they may leave the customer's system in an inconsistent state. The guidelines for writing control scripts are extensive and arcane.

There are ten different control scripts that may be used by Software Distributor. Each control script is used during different phases of software installation, and there are many subtle issues regarding the roles of each script. In particular, there are some very subtle issues when software is installed on a disk shared by several computers, since, in this case, some scripts are executed only on the machine that copies the bits, while other scripts are executed on every machine that uses the software.

Mkpkg provides a mechanism for automatically generating the control scripts for several customization actions, including PATH file updates, configuration file installation, removing obsolete files, adding a kernel driver or parameter, adding new users and groups, appending a fragment to a (configuration) file, and starting a daemon process. In addition, Mkpkg can automatically detect when some of these actions are required and will automatically create the necessary customization actions.

Control File	*Meaning*
checkinstall	executed before bits are installed to check that the software can be installed; no side-effects
preinstall	executed before bits are copied to system in preparation for installation and customization, e.g. save original versions of configuration files
postinstall	executed after bits are copied to system. Completes customizations that are shared by all systems using network bits.
configure	may be executed independently and should be executed on every system using the software. Does system-specific customizations, e.g., add a user.
verify	verifies that the software is properly installed and configured. No side-effects.
checkremove	executed before the bits are removed. Checks that a fileset or product may be removed.
preremove	executed before the bits are deleted. Prepares the system and the software for removal.
postremove	executed after the bits are removed. Cleans up and removes any leftover mess.
unconfigure	executed after the bits are removed. Removes (some) system-specific customizations. Not all customizations should be undone.

Table 2

Each customization task has a "work ticket" that specifies the type of task and any parameters. Each product and fileset has a list of these "work tickets" containing all its customization tasks.

5.1 Control files

There are nine control files that are used by Software Distributor to customize software installations. These control files fall into three basic categories: installation, verification, and de-installation. The installation scripts are used when the software is installed on the computer, and they should install and configure the software so that it may be removed safely in the future. The de-installation scripts are designed to remove most of the customizations added by the installation phase. This is an exceptionally difficult process to perfect, especially since some customizations should not be removed because the system may now depend upon them. For example, it might be a bad idea to remove a user since the customer may have created files using that user-id.

5.2 Control file generation

When mkpkg is creating a product or fileset, it iterates through the work tickets to create a sequence of control fragments in each control script. If a work ticket specifies a control action in that script, then it generates and returns the fragment. Otherwise, it returns an empty string.

5.3 Control actions

The basic customization actions include: adding directories to the PATH files, adding new users or groups to the system, installing configuration files, inserting fragments to system files, removing obsolete files, adding a kernel driver or parameter, starting a daemon process and adding cron actions.

I have built over three hundred software installation packages for a wide variety of public domain applications, such as database systems, editors, compilers, and games. These software packages vary widely in many ways, but they have needed only a few *types* of customization. I believe that the entire body of software that I have managed needs only those customizations that have already been automated by mkpkg.

I was able to obtain a copy of all the control scripts written for all the software shipped by Hewlett-Packard for HP-UX using Software Distributor. I examined nearly all of the control scripts, and I think that most of the customization actions needed by Hewlett-Packard are already included in this list.

5.3.1 Custom scripts

Since mkpkg only contains built-in customization detection and handling for a few common actions, mkpkg provides a mechanism for publishers to execute their own custom scripts. "Custom" scripts allow the user to specify that a given script be executed as part of a given control script phase. The given script is included in the package as a control

script. Mkpkg creates a control script fragment that executes the given script and retains the exit code.

5.3.2 PATH file components

In HP-UX 10.x, there are three files */etc/PATH*, */etc/MANPATH*, and */etc/SHLIB_PATH* that provide each user with default values for login shell environment variables. Any package that has its own directory tree would probably need to modify these files. For example, the new standard for independent software packages recommends that packages be installed under */opt/package/{bin,lib,etc,man,...}*, so each package would require adding path elements to each of the path files. Fortunately, it is possible to automatically recognize that a fileset requires control script actions using filename pattern matching and file type checks.

5.3.3 Configuration file installation

In HP-UX 10.x, configuration files should not be installed directly into place because end-user's may modify configuration files. In addition, in a network file system environment, a package may be installed on one machine into a shared directory, and then run by each of the other machines after "configuration". Consequently, the configure scripts need to copy configuration files into an unshared location. By convention, configuration files are contained under the */etc/* tree, but they may be located elsewhere. Although, the system only automatically detects configuration files in some cases, mkpkg users may packages may specify additional configuration files.

Since configuration files must be installed into a staging location, mkpkg searches for configuration files that are slated for installation directly into the configuration area, modifies their installation location to the staging area, and creates a customization work item. For files that have been slated for installation in the configuration staging area, mkpkg creates the customization work item to move the configuration file into place.

5.3.4 New users and groups

Some packages require that files be owned by a particular user or group. If it is not in the standard set of users and groups for HP-UX machines, then it must be installed on the system. For example, many database systems need to run as a specific user-id, and all of the database's files are owned by that user-id.

Mkpkg can generate the control script fragments that create and remove user-ids and group-ids.

In many cases, mkpkg can automatically detect that software requires a new user-id or group-id by examining the ownership of all the files in the product or fileset. If the software has user or group ownership by any user-id or group-id that is not in the basic set of users and groups shipped with every system, then mkpkg needs to create a new user-id or group-id.

5.3.5 System file modification

Some packages need to modify existing system files. There is a class of system files that contains system configuration information and that is relatively insensitive to the ordering or location of entries. For example, */etc/inetd.conf* contains a list of processes that should be started or stopped on entry and exit from various run-levels. Each entry is a single line, and the lines are position-insensitive.

Mkpkg generates the control scripts that can add or remove a line (or lines) to such files. Mkpkg needs to know the file name and line (or lines) to insert. During customization, the generated control scripts check the file for the specified lines. If they are not present, then they are appended to the system file. Decustomization may need to remove these lines from the file.

5.3.6 Crontab entries

The cron system is used to execute scheduled and repetitive actions. Each user may have an individual cron schedule. Cron uses a structured configuration file, called a crontab, to control its actions. The standard system file modification action would be sufficient for cron, except for the fact that the crontab should not be modified directly. One gets a copy of the current crontab file by executing "crontab -l" and then updates the crontab by executing "crontab <crontab>" as the user whose cron schedule is being updated.

5.3.7 Removing obsolete files

Often machines are used for years, so customers install upgraded versions of software on top of old versions. If files belonging to an old version of the software are no longer needed, the control scripts must remove them.

5.3.8 Update kernel driver or parameter

Some packages need to add a kernel driver or parameter to the kernel configuration. This is rare, but very difficult to get right and very costly if something goes wrong.

5.3.9 Starting a daemon process

Some software contains daemon processes. In many cases, the packages modify one or more system files so the daemons will be restarted automatically after a reboot. However, it is often useful to start these daemon processes immediately, without requiring a reboot. This task ensures that the daemon is started automatically on the end-user's machine during package customization.

6 Architecture

Mkpkg has a very modular design that provides a framework for adding new modules and functionality. The user requests that mkpkg execute actions, such as "create the product manifest". Actions are composed of sequences of operations. The operation is the basic unit of functionality.

Mkpkg executes each operation within an action in sequence. The operations may return an error code (in which case mkpkg may ask the packager if they would like to abort) and mkpkg adds text to the operation log. Operations have a uniform function interface. Operations are intended to function without user interaction, since mkpkg can be used by either a command-line interface or a GUI.

New functionality is added to the system by developing new operations, and then adding them to the appropriate action list or creating a new action.

Most of the internal structure of the system, its interaction, and user interface are all defined by data structures that specify how various pieces interact. In this way the basic code is often very simple and many pieces of the system can be reused easily and often. Sometimes the data structures are code fragments that get dynamically executed by the Tcl interpreter.

6.1 Data structures

Mkpkg's greatest weakness is its data structures for storing package configuration information; Mkpkg uses Tcl arrays as the basic data structure container. There are two global arrays: database() and product(). Database() contains the system information that is used by all packages created on that system, while product() contains the information relevant to a particular product.

The array index is a comma-separated list of defining attributes of the data value. All the context for a given piece of information is encoded in the index. For example, the list of prerequisites for mkpkg's fileset mkpkg-BIN is in the array element:

product(mkpkg,fileset,mkpkg-BIN,prerequisite)

This system is cumbersome but effective for most of mkpkg's needs. As I have been developing the control script generation, its weaknesses for general hierarchical data have become more pronounced.

6.2 Operations

Operations are the basic building blocks of mkpkg. Each operation is atomic and may be used in many actions. Operations often modify the package or mkpkg state, but not always. For example, one action creates the timestamp file used during manifest generation, while another action builds the application. Not all actions modify mkpkg's state; some are used to provide error checking.

6.3 Backends

The backends provide installation system-specific code. The backends provide two functions: dumpPSF, and package. DumpPSF creates the PSF file for the package using all the state and information available. Package executes the backend-specific packaging program to create an installation package.

Mkpkg is structured so that it can easily produce packages for a variety of software installation tools. In the past it has been able to create installation packages for several other installation tools.

6.4 Interfaces

There are two user interfaces for mkpkg: a command-line interface and a graphical user interface. The command-line interface provides access to most of the actions and functionality of mkpkg, but it does not have any facilities for browsing or modifying specific data fields.

The graphical interface provides access to all of the actions and functionality provided by mkpkg. In addition, it provides the ability to browse and edit all of the product configuration information, such as fileset manifests. Users may use the GUI to create, delete, or rename filesets and subproducts; to add or delete files from manifests; to specify customization

actions; or to edit any one of the other myriad configuration items.

7 Developing a package

We will walk through the whole process of developing a simple package using mkpkg. This simple package does not utilize or require all of mkpkg's functionality, but it does demonstrate features common to most packages.

Our first step is to prepare the software for packaging. We should be able to automatically compile and install the software correctly on our machine without human intervention. Of course, if we are building a package for pre-compiled software, we can skip compilation. In our case, we have a Makefile with three targets: *all*, *install*, and *clean*.

Our software sources are in *less-1.0*, and our package contains the following files:

/usr/local/bin/less
/usr/local/bin/X11/xless
/usr/local/lib/X11/app-defaults/Xless
/usr/local/man/man1/less.1
/usr/local/man/man1/xless.1

Table 3

Since the package is small, and since it has only a few man pages, we will ship the entire product in a single fileset *less-RUN*.

We need to decide if we will distribute code that has been statically linked or dynamically linked. In general, software that is released as part of HP-UX will be dynamically linked, while software that is shipped by third parties or is shipped independently of HP-UX may be statically linked. The advantage of static linking is that the executables do not depend on specific shared libraries and are more likely to work correctly on a wider range of platforms, but at the cost of additional disk space consumption. Our package will be shipped with dynamically linked executables. We are now ready to begin building our Software Distributor (SD-UX) product.

Secondly, we start mkpkg within our project directory *less-1.0*, and provide mkpkg with enough information to be able to build, install, and locate the software in our product. The mkpkg interface has a menu bar across the top. Under the 'View' menu, we can see all of the 'pages' that contain information that we may need to provide, verify, or modify. Under the 'Action' menu are all the actions that we need to produce an installation package.

We are currently viewing the 'configuration' page for the product. Notice that mkpkg has already provided default values for many of the attributes. Some of the defaults come from the default values on the 'system' pages, but others have been computed. For example, the product name (less) and version (1.0) have been computed from the current directory name.

On the configuration page, we need to fill in the 'directory' attribute with */usr/local*. This attribute is used in the PSF file, but it is also used by mkpkg during manifest generation to locate the files installed as part of the package. In some cases, we may not know where every file will be installed. Mkpkg has a (long) list of directories in order to catch these wayward files.

We have told mkpkg where to look for installed files, now we need to tell it how to build and install our software. Go to the 'build' page under the 'View' menu and check the attributes 'build', 'install', and 'clean'. Their defaults 'make', 'make install', and 'make clean' are correct because they are the targets used by our Makefile.

We need to create the manifest, so select the menu option 'Create file list' on the 'Action' menu. This action may take a long time, since it compiles and installs the software, and then it searches your system for newly installed files. For large packages, just compiling the software may take hours, while for large systems it may take hours to just search the file system for installed files. Once this action is complete, you should see a fileset 'less-RUN' and a subproduct 'Runtime' under the 'View' menu.

You should now check every attribute on each page, correcting or providing information as necessary. You should also check that the fileset *'less-RUN'* contains all the files from our package (and no more!), and that the subproduct *'Runtime'* contains just one fileset. Also, *'less-RUN'* should have dependencies on various filesets from OS-CORE and X11[1].

Thirdly, we create the installation package with the 'Create dynamic package' action. This action compiles and installs the software. It then copies the software to some 'safe' place (in the parent directory

[1] Actually, mkpkg will only find these shared library dependencies if you have run the action 'Search system for shared libraries' prior to running 'Create file list'.

of *less-1.0*, there should be a directory named something *like less-1.0__10.20__dynamic*). It then generates a PSF file (in the 'safe' place), and uses that PSF to create an installation package. The installation package is left in the parent directory.

8 Experiences

I started developing software installation packages in 1993 because I wanted to provide binary installation for public domain software within Hewlett-Packard. There is a network installation tool, called *ninstall*, which has been in widespread use within Hewlett-Packard for a long time. I wanted to build *ninstall* packages for common public domain software, such as emacs, so other people inside Hewlett-Packard could install the packages and not duplicate my porting, configuration, and compilation effort.

It took me a week to generate my first *ninstall* package, both because I had to learn how to package software and because I had to manually create the package manifest and all the PSFs. It only took a week to write the first version of mkpkg, which included the automatic manifest generation and PSF generation.

I used the initial version to develop binary installation packages for about 50 packages. At this point it would take me about three minutes of effort per-package to build a complete binary installation package once the software had been ported and configured. Since mkpkg builds the package several times during the course of the process, the actual elapsed time can be far longer.

At this point I was supporting a library of about 50 public domain software packages which could be installed by HP employees over the HP intranet. This library was very popular and I soon had thousands of internal "customers"; I also started getting bug reports. I discovered that my customers were having a lot of problems running programs that depended on a shared library that was missing on their machine. Usually the library was included in another package, but I had not marked the package dependency so the requisite libraries were not getting installed automatically. I then extended mkpkg to detect and manage shared library dependencies and the problems disappeared.

Using this version of mkpkg, I have been supporting over 250 packages. Once the software is ported, debugged, and configured, I can usually generate a binary installation package with about three minutes of effort. The biggest difficulty at this stage was developing customize/decustomize scripts for extraordinary packages. In addition, I found a few packages (e.g. TeX 3.1415) whose "`make install`" processes were so intelligent that the processes would only install certain files if they did not already exist. Since these files invariably existed on my machine, they were not installed during the "make install" phase of manifest generation and so they were not included in the manifest. There is no substitute for testing software packaging on a "clean" machine.

Mkpkg was then extended so that it could generate both *ninstall* and *update* packages. This version of mkpkg was shared with the HP-UX Porting and Archive Centre so they could easily generate *update* packages of public domain software.

With the advent of HP-UX 10.0, *update* was replaced with SD-UX, the HP-UX version of Software Distributor, as the standard software installation tool. Since SD-UX added hierarchical structure on top of the simple fileset model used by *update*, mkpkg was rewritten to manage the product/subproduct/fileset structure. This hierarchy added a lot of complexity to mkpkg. For example, mkpkg now knows how to create filesets and subproducts based on standard naming conventions and other guidelines, and during manifest generation it automatically assigns files to the proper fileset. Internally, Hewlett-Packard has a variety of guidelines governing subproduct and fileset naming, assignment of files to filesets, and a myriad of other topics, and mkpkg tries to automate those guidelines whenever possible.

This hierarchical version of mkpkg was also shared with the HP-UX Porting and Archive Centre so they could start generating SD-UX packages. They have since used it to generate thousands of packages.

My final task was developing customize/decustomize scripts. While developing hundreds of *ninstall* packages I discovered that most packages require only a few, basic customization actions, so mkpkg was extended to automatically detect and generate customize/decustomize scripts for a variety of common actions.

9 Acknowledgements

I would like to thank Colin Charlton, Richard Lloyd, Rik Turnbull, and all of the dedicated people of the HP-UX Porting and Archive Centre who have put up with pre-release versions of mkpkg and provided valuable feedback.

I would also like to thank Shahryar Shahsavari of Hewlett-Packard Software Integration and Distribution Organization who gave me a great deal of advice and information while I was designing the automated customization script generation.

Finally, I should like to thank David Mullaney, George Williams, Mark Mayotte, Debbie Ogden and the rest of the Software Distributor team for their support and encouragement.

10 Conclusions

Mkpkg dramatically simplifies the process of creating installation packages, by automating most parts of the software package creation process. Using mkpkg a skilled user can create complex binary installation packages for Software Distributor in a few minutes of effort, a process which used to take hours or days.

Binary installation packages are very useful, but they have primarily developed and distributed by large software and operating system vendors because they are so difficult to develop. By dramatically reducing the effort and complexity associated with developing binary installation packages, it should now be possible for harried system administrators and MIS support staff to develop their own binary installation packages for software that they support and redistribute with their organization.

11 Bibliography

[1] *Managing HP-UX Software with SD-UX.* Hewlett-Packard. Part Number B2355-90080. 1995.

[2] Edward Bailey, *Maximum RPM: Taking the Red Hat Package Manager to the Limit*, Red Hat Software, Durham, North Carolina, 1997.

[3] *RPM HOW-TO*, http://www.rpm.org/support/RPM-HOWTO.html

[4] *Software Distribution Standard*, The HP-UX Porting and Archive Centre, http://hpux.csc.liv.ac.uk/hppd/standard.html

[5] Standard for Information Technology – Portable Operating System Interface (POSIX) System Administration. IEEE POSIX draft P1387.2/D13, April 1994.

[6] Scott Hazen Mueller, *Good programs, lousy installation.* ;login: (21)3:36-38, USENIX. June 1996.

Glossary

Bundle
A collection of products and filesets that are installed as a unit by Software Distributor.

Control script
A script that is contained in a product or fileset and which is used by Software Distributor to check or modify the system state during software installation or de-installation.

Cron
A UNIX service that executes system and user-defined actions periodically. Its configuration file is *crontab*.

Crontab
The configuration file used to control *cron*. It may not be edited in place; rather, one must extract the current *crontab* using the command 'crontab -l' and then install a new crontab using the command 'crontab'.

Customization Actions
Actions that are not standard and that occur during software installation and de-installation. Mkpkg customization typically modifies the system configuration so that the software runs correctly. These actions occur without user interaction.

Dependencies
An attribute of a package that indicates whether the package requires another package to work properly. Dependencies may be either 'prerequisite' (if the package must be installed before the current package is installed) or 'corequisite' (if the package must be installed before the current package is executed).

Dynamic linking
In UNIX, executables can be linked to shared libraries using 'dynamic linking', which means that the executable does not contain a copy of the library, but only a reference to the library. If the library changes, then the executable may no longer work properly.

See also **Static linking**

Fileset
The atomic unit of installation. Contains files and customization scripts (if applicable). May also have additional requirements, such as dependencies.

Man page
Manual page for UNIX's online documentation system.

Manifest
The list of all files to be installed.

Product
The primary unit of software installation in Software Distributor. It contains both subproducts and filesets and may have installation dependencies and customization actions.

Product Specification File (PSF)
The file that describes the entire product or bundle.

SD-UX
The HP-UX version of Software Distributor, which is the standard software installation tool provided for HP-UX. All Hewlett-Packard software for HP-UX is distributed in this format.

Software Distributor
A POSIX-1003.7.2-compliant software installation toolset developed by Hewlett-Packard.

Static linking
With static linking, executables contain a copy of the various library routines rather than references to shared libraries. These executables are often significantly larger that dynamically linked executables, but they are more likely to work on a variety of platforms and are less fragile in the face of operating system upgrades.

See also **Dynamic linking**

Subproduct
A collection of filesets that are installed as a unit by Software Distributor. Subproducts are contained in products, and a product can have several subproducts. Filesets may be contained in more than one subproduct.

IP.com PriorArtDatabase

Technical Disclosure

Archived Kernel Extensions

Author(s): **IBM TDB**
Mealey, BG
Swanberg, RC

IP.com number: **IPCOM000123901D**
Original Publication Date: **June 1, 1999**
Original Disclosure Information: **RD v42 n422 06-99 article 422131**
IP.com Electronic Publication: **April 5, 2005**

IP.com, Inc. is the world's leader in defensive publications. The largest and most innovative companies publish their technical disclosures into the IP.com Prior Art Database. Disclosures can be published in any language, and they are searchable in those languages online. Unique identifiers indicate documents containing chemical structures as well as publications open for comment in the IP Discussion Forum. Disclosures are published every day online and also appear in the printed IP.com Journal. The IP.com Prior Art Database is freely available to search by patent examiners throughout the world.

Client may copy any content obtained through the site for Client's individual, noncommercial internal use only. Client agrees not to otherwise copy, change, upload, transmit, sell, publish, commercially exploit, modify, create derivative works or distribute any content available through the site.

Note: This is a pdf rendering of the actual disclosure. To access the notarized disclosure package containing an exact copy of the publication in its original format as well as any attached files, please download the full document from the IP.com Prior Art Database at: http://www.ip.com/pubview/IPCOM000123901D

www.ip.com

Copyright © IP.com, Inc. All rights reserved.

Archived Kernel Extensions

Disclosed is a method for managing multiple versions of a kernel extension or device driver for a single system. The method employs existing technologies utilized in archival of loadable shared objects, applied at the kernel level for loadable kernel images.

Supporting multiple versions of the same kernel extension or device driver in a single installed system image presents several difficulties to installation, system administration, and update procedures. For example, with the evolution of today's hardware systems, 64-bit technology has required operating systems to now provide 64-bit kernels. The 64-bit kernels have in turn required kernel extensions and device drivers to then become 64-bit, all the while continuing to support their 32-bit predecessors. A further complication is the ability for some processors, such as the *PowerPC, to run either a 32-bit or 64-bit environment. This gives the customer the flexibility to choose to run a 32-bit operating system kernel or a 64-bit operating system kernel depending on specific scalability and application workload demands. With this flexibility comes the requirement that at least two versions of every kernel extension and device driver be available for selection dependent on the operating environment chosen.

Using traditional kernel extension and device driver packaging and configuration methods, this would have required another complete set of kernel extension and device driver installable file sets to be created and maintained. Unique extension names would have been introduced to distinguish the 64-bit versions from the 32-bit versions, also impacting all areas where an extension or driver is known by name. Much of the content of these file sets, with the exception of the extension binaries themselves would be identical, but duplicated. Also, when switching between 32-bit and 64-bit operating environments, somehow the appropriate set of kernel extensions and device drivers would need to be activated over the other.

The disclosed solution solves each of these problems by maintaining a single name space per kernel extension and device driver. The solution is to package each kernel extension and device driver in an archived file format which contains multiple versions of the extension or driver. Specifically in the case of 64-bit versus 32-bit operating environments, the archived file would contain the 64-bit extension binary and the 32-bit extension binary. The kernel loader is then enhanced to recognize the archived file format when loading the extension, perform a run-time check to determine the current operating environment and then conditionally load the correct kernel extension member from the archive for the selected environment. This allows all entities referencing the extension or driver by name to remain unchanged. Thus major functions like system installation, system administration, service update, and switching between 32-bit and 64-bit operating environments are not impacted by the existence of multiple kernel extension or device driver versions present on the system.

*PowerPC is a trademark of International Business Machines Corp.

Disclosed by International Business Machines Corporation

IP.com PriorArtDatabase

Technical Disclosure

Design of Application Install Plan Object for Network Installation

Author(s): **IBM TDB**
Bunce, JL
Clark, KA
Shrader, TJL

IP.com number: **IPCOM000116613D**
Original Publication Date: **October 1, 1995**
Original Disclosure Information: **TDB v38 n10 10-95 p187-188**
IP.com Electronic Publication: **March 31, 2005**

IP.com, Inc. is the world's leader in defensive publications. The largest and most innovative companies publish their technical disclosures into the IP.com Prior Art Database. Disclosures can be published in any language, and they are searchable in those languages online. Unique identifiers indicate documents containing chemical structures as well as publications open for comment in the IP Discussion Forum. Disclosures are published every day online and also appear in the printed IP.com Journal. The IP.com Prior Art Database is freely available to search by patent examiners throughout the world.

Client may copy any content obtained through the site for Client's individual, noncommercial internal use only. Client agrees not to otherwise copy, change, upload, transmit, sell, publish, commercially exploit, modify, create derivative works or distribute any content available through the site.

Note: This is a pdf rendering of the actual disclosure. To access the notarized disclosure package containing an exact copy of the publication in its original format as well as any attached files, please download the full document from the IP.com Prior Art Database at: http://www.ip.com/pubview/IPCOM000116613D

ip.com

www.ip.com

Copyright © IP.com, Inc. All rights reserved.

AT894-0762 JSL
J. L. Bunce, K. A. Clark and T. J. L. Shrader

IBM® Technical Disclosure Bulletin Vol. 38 No. 10 October 1995

Design of Application Install Plan Object for Network Installation

Disclosed is a design for creating application install plan objects that act as snapshots of applications and their attributes which the user specified. The burdens on network administrators have been rapidly growing both in volume and in complexity. Chief among them is the need for administrators to easily plan and execute the installation and configuration of software products on a group of workstation on a LAN. The design of the application install plan object can be used with any network planning, installation, and configuration program.

The Application install plan (AppIP) object is the install plan counterpart to the application object. (An install plan object collects workstations and the applications to be installed on the workstations.) An application or AppIP object represents a software product like OS/2* or LAN Server*. AppIPs differ from application objects because they are snapshots of the application's attributes when the application is added to the install plan. An AppIP object is created when an application object is added to the install plan. The AppIP object inherits all attributes from the application from which it was created. The user cannot change the application short name, application backing file name, or the application action type (such as install or configure) in the AppIP object.

When an application is added to an install plan, the network installation application could check to see if the "prompt user for action type" checkbox was checked. If so, the network installation application could post a dialog with the action types the user can select from, such as install or configure. This action type will be stored in the AppIP. If the prompt user for action type checkbox was not checked, the network installation application could take copy the action type setting the application object to the AppIP object. This action allows users the flexibility to use a default action type or choose an action, like install or configure, when the application is added to an install plan object.

An AppIP can have 0 to N CatIP objects and 0 to 1 customization file objects. Category Install Plan (CatIP) objects group together similar CID response files, such as all install response files. Customization files allow users to modify values of response file keywords for a number of workstations. A valid AppIP does not have to contain any objects since some applications can be installed or configured without response files or user interaction.

In other designs, users do not have the flexibility to have application type objects which differ both outside an install plan or between install plans.

This design allows users to take snapshots of applications and customize them per install plan. The design is flexible to allow action types to be changed when an application is added to an install plan.

* Trademark of IBM Corp.

Mkpkg: A Software Packaging Tool

Carl Staelin
Strategic Planning and Communications
HPL-97-125 (R.1)
August, 1998

software packaging,
software distribution,
software publishing

Mkpkg is a tool that helps software publishers create installation packages. Given software that is ready for distribution, mkpkg helps the publisher develop a description of the software package, including manifests, dependencies, and post-install customizations. Mkpkg automates many of the painstaking tasks required of the publisher, such as determining the complete package manifest and dependencies of the executables on shared libraries. Using mkpkg, a publisher can generate software packages for complex software such as TeX with only three minutes effort.

Mkpkg has been implemented on HP-UX using Tcl/Tk and provides both a graphical and command line interface. It builds product-level packages for Software Distributor (SD-UX).

© Copyright Hewlett-Packard Company 1998

Mkpkg: *A software packaging tool*

Carl Staelin

January 14, 1997

Abstract

Mkpkg is a tool that helps software publishers create installation packages. Given software that is ready for distribution, mkpkg helps the publisher develop a description of the software package, including manifests, dependencies, and post-install customizations. Mkpkg automates many of the painstaking tasks required of the publisher, such as determining the complete package manifest and dependencies of the executables on shared libraries. Using mkpkg, a publisher can generate software packages for complex software such as TeX with only three minutes effort.

Mkpkg has been implemented on HP-UX using Tcl/Tk and provides both a graphical and command line interface. It builds product-level packages for Software Distributor (SD-UX).

1 Introduction

Most end-users do not build programs from source code, but install software using binary installation packages. Mkpkg helps software publishers develop those installation packages.

Mkpkg addresses a part of the software distribution channel that has been largely ignored. Most software distribution systems have focussed on defining the binary package format and the protocols for installing and de-installing software. Most software installation suites have made it very easy for end-users and system administrators to distribute and install software, but they have not addressed the problems of the software packager who is creating binary installation packages.

The software publishing process includes several actors and steps: the software developer, the software publisher, the distributor, (sometimes the system administrator), and the end-user. The software developer creates the software. The packager is responsible for configuring, compiling, and packaging the software to create the binary installation package that the distributor delivers to the end-user. In some environments system administrators install and manage the software for end-users.

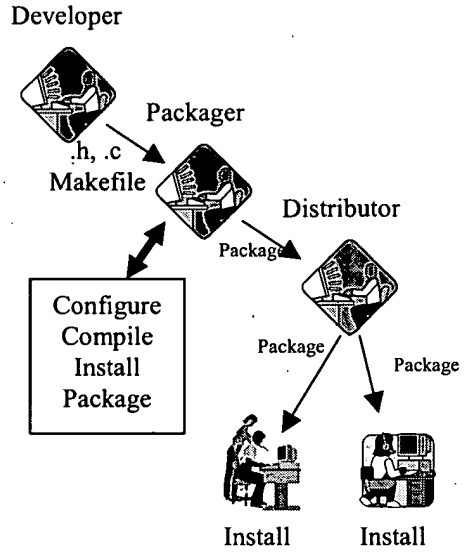

Figure 1

Mkpkg helps software packagers create installation packages. Typically, the packager starts with source code that needs to be compiled and installed on the packager's computer. The packager tries to create an installation package that re-creates the installation on each end-user's computer.

Developing a binary software installation package, that is, creating a package that can be installed easily on a computers and have it work properly, is an important and difficult task. Most vendors have developed tools that can accept the descriptions of a software package and create an installation package, but developing those package descriptions is difficult. Package descriptions typically contain the following elements:

Each software installation tool has its own idiosyncrasies and requirements, but they all share these common elements. Many software installation tools also provide other elements, such as system specifications that define which hardware/OS types or versions may install the software.

Element	Description
title	Software package name
description	A text description of the package and its capabilities
manifest	A list of all the files contained in the package
dependencies	A list of all the other packages required for this package to operate correctly
customization scripts	A set of scripts that are executed on the user's machine during installation or de-installation of the software

Table 1

1.1 Software Distributor

Software Distributor (SD-UX) is a suite of software installation programs that satisfy the POSIX draft 1387.2 specification. It is the software distribution mechanism for all Hewlett-Packard software for HP-UX and has versions that run on at least WindowsNT and Solaris.

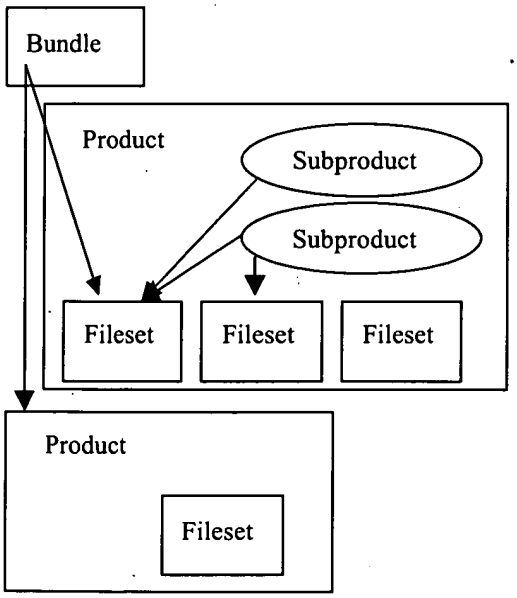

Figure 2

Software Distributor has four levels of software grouping: bundle, product, subproduct, and fileset. A bundle is a collection of products and/or filesets that may be installed as a unit. Bundles were designed to provide customers with one single installation unit for purchased software products, such as the ANSI/C compiler. Bundles may be used to provide a logical grouping by function, such as "web server".

The basic unit of software distribution is the product. A product may contain both subproducts and filesets. Subproducts contain filesets and are used to manage logical subsets of a single product.

Filesets are the atomic units of software distribution and contain a set of files and control scripts. SD-UX installs and configures individual filesets; filesets cannot be partially installed or configured.

Software Distributor has two levels of software distribution: bundle and product. The basic unit of distribution is the product. Software Distributor has several levels of software installation. The basic unit of installation is the fileset, but customers usually install software at the bundle or subproduct level.

Mkpkg creates product packages, while mkbdl creates bundles. Since all filesets and subproducts are created as part of a product, we do not provide a separate tool for creating them.

1.2 Installation tools

There are many methods for distributing binary installations. Each method has various strengths and weaknesses, but most commercial systems provide a similar level of basic operation. The largest UNIX vendors have each developed their own systems for distributing binary software: HP's HP-UX uses Software Distributor, Sun's Solaris uses *pkgadd*, Digital's OSF/1 uses *setld*, SGI's IRIX uses *inst*, and Linux uses RPM. Windows has two standards, InstallShield packages and self-extracting programs.

Each of the commercial systems offers basic services, such as installing and deinstalling packages atomically, tracking and managing inter-package dependencies, and executing scripts during software installation and deinstallation. Most of them also support a variety of more advanced features, such as versions, operating system and hardware dependencies, and interactive installation. For the most part these systems try to make it as easy as possible for system administrators to install software.

1.2.1 Tar

The simplest binary installation package is simply a tar file containing the software. Often such packages include a README file that includes installation instructions. For simple programs and packages, tar files are often sufficient. For more complex packages, much of the burden of correctly installing

and configuring the software falls on the end user because the installation process for tar files is completely manual.

Occasionally, software publishers will include installation scripts as part of the tar file and the installation script will automatically install and configure the software for the user. One of the few publishers who uses this approach is Netscape, who includes an "ns-install" script as part of their Netscape Communicator tar-file distribution.

1.2.2 RPM

The RedHat Package Manager (RPM) [2,3] was developed for the Linux environment and provides a very nice environment for installing and distributing software. Functionally, it is very similar to Software Distributor; it includes support for inter-package dependencies and control scripts that are executed during software installation.

Users may install software from a local depot or they may install from a remote server over the network. RPM is able to contain binaries for multiple platforms within a single package, and it can automatically install the correct binaries. Using RPM customers may determine which package installed a particular file, and what software is installed on the machine. Users may also uninstall packages.

RPM has some support for the packager, but it is missing some important features. RPM does not help the publisher develop the package manifest. The RPM documentation [3] states: "RPM has no way to know what binaries get installed as part of `make install`. There is *NO* way to do this. Some have suggested doing a `find` before and after the package install. With a multi-user system, this is unacceptable as other files may be created during a package building process that have nothing to do with the package itself."

1.3 Software configuration

Software configuration is one of the dirty little secrets of system administration. Software that is well configured works well in a broad variety of system configurations and causes few problems for system administrators. Poorly configured software can cause system administrators a great deal of aggravation.

Sometimes the difference between well-configured software and poorly configured software is a matter of tiny details, but there are a few general guidelines.

- Never compile paths into binaries.
- Separate executables, configuration files, and data or log files.
- Use human readable ASCII files for configuration information.
- Follow standard conventions for file and path names as much as possible.

System administrators frequently share file systems between systems, so executables and libraries will often be shared by many systems. However, administrators usually want each computer to have its own version of configuration files, but these may be on a read-only file system. In addition, data and log files are usually not shared between systems and usually must be mounted with read-write permissions. Software configurations must anticipate these kinds of configuration issues.

A fairly detailed set of configuration guidelines is published by the HP-UX Porting and Archive Centre [4].

2 Software packaging process

During software packaging, the publisher must prepare all the elements needed by the installation package. For many small packages, this is a very simple process, but for larger packages it can be quite difficult. Mkpkg provides five services during the packaging process:

- Creates the manifest, the list of files to be installed
- Determines the dependencies of this package on other packages
- Develops the install/de-install scripts
- Gathers all the components, as listed in the manifest
- Assembles and produces the completed installation package

2.1 Create manifest

The manifest is a list of all the files to be installed by the package. Mkpkg can automatically determine which files were installed by the package on the publisher's machine. For small packages it is easy to determine which files belong to a given package, but manual techniques often miss files and make mistakes. For larger packages, such as X11R6 or TeX, it is usually difficult to identify all the files

IP.com PriorArtDatabase

Technical Disclosure

Design of Application Install Plan Object for Network Installation

Author(s): **IBM TDB**
Bunce, JL
Clark, KA
Shrader, TJL

IP.com number: **IPCOM000116613D**
Original Publication Date: **October 1, 1995**
Original Disclosure Information: **TDB v38 n10 10-95 p187-188**
IP.com Electronic Publication: **March 31, 2005**

IP.com, Inc. is the world's leader in defensive publications. The largest and most innovative companies publish their technical disclosures into the IP.com Prior Art Database. Disclosures can be published in any language, and they are searchable in those languages online. Unique identifiers indicate documents containing chemical structures as well as publications open for comment in the IP Discussion Forum. Disclosures are published every day online and also appear in the printed IP.com Journal. The IP.com Prior Art Database is freely available to search by patent examiners throughout the world.

Client may copy any content obtained through the site for Client's individual, noncommercial internal use only. Client agrees not to otherwise copy, change, upload, transmit, sell, publish, commercially exploit, modify, create derivative works or distribute any content available through the site.

Note: This is a pdf rendering of the actual disclosure. To access the notarized disclosure package containing an exact copy of the publication in its original format as well as any attached files, please download the full document from the IP.com Prior Art Database at: http://www.ip.com/pubview/IPCOM000116613D

ip.com

www.ip.com

Copyright © IP.com, Inc. All rights reserved.

AT894-0762 JSL
J. L. Bunce, K. A. Clark and T. J. L. Shrader

IBM® Technical Disclosure Bulletin Vol. 38 No. 10 October 1995

Design of Application Install Plan Object for Network Installation

Disclosed is a design for creating application install plan objects that act as snapshots of applications and their attributes which the user specified. The burdens on network administrators have been rapidly growing both in volume and in complexity. Chief among them is the need for administrators to easily plan and execute the installation and configuration of software products on a group of workstation on a LAN. The design of the application install plan object can be used with any network planning, installation, and configuration program.

The Application install plan (AppIP) object is the install plan counterpart to the application object. (An install plan object collects workstations and the applications to be installed on the workstations.) An application or AppIP object represents a software product like OS/2* or LAN Server*. AppIPs differ from application objects because they are snapshots of the application's attributes when the application is added to the install plan. An AppIP object is created when an application object is added to the install plan. The AppIP object inherits all attributes from the application from which it was created. The user cannot change the application short name, application backing file name, or the application action type (such as install or configure) in the AppIP object.

When an application is added to an install plan, the network installation application could check to see if the "prompt user for action type" checkbox was checked. If so, the network installation application could post a dialog with the action types the user can select from, such as install or configure. This action type will be stored in the AppIP. If the prompt user for action type checkbox was not checked, the network installation application could take copy the action type setting the application object to the AppIP object. This action allows users the flexibility to use a default action type or choose an action, like install or configure, when the application is added to an install plan object.

An AppIP can have 0 to N CatIP objects and 0 to 1 customization file objects. Category Install Plan (CatIP) objects group together similar CID response files, such as all install response files. Customization files allow users to modify values of response file keywords for a number of workstations. A valid AppIP does not have to contain any objects since some applications can be installed or configured without response files or user interaction.

In other designs, users do not have the flexibility to have application type objects which differ both outside an install plan or between install plans.

This design allows users to take snapshots of applications and customize them per install plan. The design is flexible to allow action types to be changed when an application is added to an install plan.

* Trademark of IBM Corp.

UMDF Contents

For UMDF, the native WDF contains .sys, .dll, and .exe files because UMDF has both user-mode and kernel-mode components. The following are the UMDF files:

- WUDFRd.sys, which is the kernel-mode reflector
- WUDFHost.exe, which is the executable file for the driver host process
- WUDFSvc.dll, which contains the driver manager
- WUDFPf.dll, which contains run-time support for the framework
- WUDFx.dll, which contains the framework itself
- WUDFCoInstaller.dll, which is a required native co-installer

The native UMDF is available only in Windows Vista. In Windows XP, UMDF is available only through the redistributable co-installer. If the native UMDF requires critical security fixes, Microsoft will release the updated version on Windows Update. UMDF is not currently supported for Windows 2000 or Windows Server 2003.

The redistributable WDF package for UMDF is provided in a co-installer file that is named WudfUpdate_*MMmmm*.dll, where *MM* is the major UMDF version number and *mmm* is the minor version number. The redistributable package contains as resources all the files in the native UMDF. The redistributable co-installer is not supplied with the operating system or on Windows Update. Vendors acquire it in the WDK and in GDRs.

Framework and Driver Installation

All WDF drivers are installed by INF files. Each INF file must contain a [*DDInstall*.**CoInstallers**] section that installs the co-installer. The co-installer in turn reads the *DDInstall*.**Wdf** section of the INF, which provides the required information to install KMDF or UMDF:

- For KMDF drivers, the *DDInstall*.**Wdf** section contains the **KmdfService** directive, which assigns a name to the driver's kernel-mode service and points to the *Wdf-install-section*, which specifies the minimum major and minor KMDF library version that the driver requires.
- For UMDF drivers, the *DDInstall*.**Wdf** section contains the **UmdfService** directive, which assigns a name to the driver and points to the *Umdf-install-section*. The *Umdf-install-section*, in turn, specifies the minimum major and minor UMDF version that the driver requires along with the driver class ID. The *DDInstall*.**Wdf** section also specifies the order in which the UMDF drivers should be installed in the device stack and the maximum impersonation level that the UMDF driver can use.

The version of the co-installer in the driver package must match the version of the framework library that is listed in the INF file. For example, if the INF file specifies version 1.5, the co-installer version must be 1.5. Therefore, if Microsoft releases a new minor version of the co-installer (and thus of the library itself), vendors that use the new co-installer must revise their INF files to specify the new co-installer and framework. Because the INF file is signed, such changes require a new signature. Alternatively, vendors can continue to use the older minor version and end users can get the newer minor framework version from Windows Update.

© 2006 Microsoft Corporation. All rights reserved.

The co-installers and the resources that they contain are all signed components. Driver installation fails if the certificate with which the co-installer was signed is not available on the target system. The co-installers record information in the system event log. When debugging installation problems, vendors should check the following event logs for relevant details:

- The WDF installation log (%windir%\wdf*MMmmm*inst.log, where *MMmmm* indicates the major and minor version numbers) contains information about events and errors that occur during installation.
- The Setup action log (%windir%\setupact.log) contains debugging messages from the WDF co-installer.
- The UMDF update log (%WINDIR%\temp\wudf_update.log) contains messages from the UMDF update.exe package.
- System event log, available through the Event Viewer, contains information about errors that occur during dynamic binding of a KMDF driver to the library.

Binding to the Framework

WDF drivers bind dynamically with the framework at load time. This section describes how the binding occurs.

Dynamic Binding for KMDF

At build time, KMDF drivers link statically with WdfDriverEntry.lib. This library contains information about the KMDF version in a static data structure that becomes part of the driver binary. The internal FxDriverEntry function in WdfDriverEntry.lib wraps the driver's **DriverEntry** routine, so that when the driver is loaded, FxDriverEntry becomes the driver's entry point. At load time, the following occurs:

1. FxDriverEntry calls the KMDF loader (which is defined in wdfldr.sys) and passes the version number of the KMDF framework library with which to bind.
2. The loader determines whether the specified major version of the framework library is already loaded. If the specified version is not yet loaded, the loader starts the KMDF run-time component. If the run-time component starts successfully, the loader adds the driver as a client of the service and returns the relevant information to the FxDriverEntry function. If the driver requires a newer version of the run-time library than the one already loaded, the loader fails and logs the failed attempt in the system event log.

Binding for UMDF

The UMDF framework runs in the driver host process along with the vendor's user-mode driver code and the Microsoft-supplied run-time environment. The framework is a set of object interfaces that are exposed as COM interfaces and packaged in a DLL. The UMDF driver is also a COM component that controls the hardware from user mode. The run-time environment handles I/O dispatching, driver loading, driver layering, and the thread pool, and communicates with the kernel-mode drivers for the device.

At build time, every UMDF driver includes the Microsoft-supplied header file wudfddi.h, which exports information that indicates which version of UMDF the driver requires. When the driver is loaded, the UMDF run-time environment checks this exported value to ensure that the required version number is less than or equal

© 2006 Microsoft Corporation. All rights reserved.

to the currently installed version. If the driver requires a newer version than the one already installed, the loader fails.

Versioning Scenarios

The following are common situations that involve versioning.

End user installs a driver that is supplied with a device. When the end user installs a driver that is supplied with a device, installation proceeds as follows:

- If the framework is not already installed on the system, the co-installer that is distributed with the driver installs the framework from the distribution medium. For example, this situation occurs the first time a user installs a WDF driver on Windows XP.
- If the framework is already installed on the system, the co-installer installs the framework from the distribution medium only if it is newer than the installed version.
- For UMDF, if the framework is already installed but one or more files are missing, the co-installer installs the framework from the distribution medium if it is the same as or newer than the installed version.

At installation, a new minor release overwrites the older existing minor release. Thus a system can have at most one minor release of each major release at any one time. UMDF supports rollback to an earlier minor version through the Add or Remove Programs application in Control Panel. The KMDF co-installer does not currently support rollback.

Example 1: Framework version 1.4 is present on the system. The user installs a driver that is packaged with the co-installer for framework version 1.2. Because the existing minor version is more recent, the co-installer does not install the framework and the new driver is linked to framework version 1.4.

Example 2: Framework version 1.4 is present on the system. The user installs a driver that is packaged with the co-installer for framework version 1.5. Because the existing framework is older than the one in the driver package, the co-installer installs version 1.5 and both the new driver and all existing drivers that require major version 1 are linked to version 1.5.

- For KMDF, if version 1.4 is already loaded in memory when the user installs the new driver, the user must reboot the system so that the existing drivers can relink against version 1.5.
- For UMDF, if existing devices are already using version 1.4, the co-installer temporarily stops all of the UMDF devices on the system, installs version 1.5, and then restarts all the devices. If a driver for any of the devices vetoes the query-stop request, the co-installer prompts the user to reboot the system.

Version 1.4 is deleted from the system.

End user installs a device that has a Windows-supplied driver. The end user attaches a device for which Windows includes a driver. If this is the first WDF device driver to be activated on the system, the driver is linked to the appropriate framework version. Drivers that are supplied with Windows must be compatible with the native WDF that is included with the operating system release.

© 2006 Microsoft Corporation. All rights reserved.

Microsoft releases a new minor version of the framework. Microsoft finds and fixes a bug in the driver framework and releases a new minor version. The new minor version is made available to vendors through a GDR to distribute with their devices and to end users on Windows Update. Vendors can either:

- Update their driver packages to use the new co-installer. Such an update requires changes to the INF and thus a new signature.
- Continue to use the old co-installer. End users can get the updated WDF from Windows Update. However, the vendor should nevertheless test the driver to ensure that it continues to work properly with the new minor version.

If the co-installer itself contains a critical bug, such as one affecting system security, Microsoft releases the updated co-installer on Windows Update.

Hardware vendor updates a driver. A hardware vendor fixes a bug in a driver and makes the updated driver available on the vendor's Web site. If a new minor version of the framework has been released since the original driver was published, the vendor might decide to include the co-installer for the new minor version along with the updated driver. The end user downloads the new driver installation package, uninstalls the old driver, and installs the new driver. If the framework in the driver installation package is newer than the one that is already on the user's machine, the co-installer updates the framework on the user's machine.

Developer installs a device on a machine with a prerelease WDF build. This scenario can occur only in a development environment—not on an end user system—because the beta releases of the co-installers are not redistributable.

A driver developer installs a new device that uses the final, released version of the framework. The developer has previously installed a prerelease build of the same version number on the machine. The installation procedure does not overwrite the prerelease build of the co-installer; instead, the driver developer must delete the prerelease co-installer manually before installing the new device.

For example, Microsoft releases the framework version 1.5. A driver developer has already installed the beta build of version 1.5 on the development system. The developer then tries to install a new device that is packaged with the final version of the framework. Because the two co-installers have the same name, Setup uses the version that is already in the System32 directory on the machine. To ensure that the final release installs correctly with the device, the driver developer must first delete the beta release of the co-installer and then install the device. After the beta co-installer has been deleted, installation can proceed to use the correct co-installer to update the framework.

Currently, the co-installers for UMDF and KMDF do not run on prerelease versions of Windows Vista because Windows Vista includes native versions of both frameworks. Updated prereleases of KMDF and UMDF for Windows Vista are available only with prereleases of the operating system. When the developer installs a newer version of Windows Vista, the frameworks will be updated.

Best Practices for Vendors

The following are some best practices for vendors:

- Ensure that you build the driver against the same version of the framework as the co-installer that you supply on the installation media.
- When Microsoft releases a new version of the framework, always perform regression testing with your driver. By supporting the latest framework, you

© 2006 Microsoft Corporation. All rights reserved.

ensure that your driver benefits from new features and bug fixes and help to provide a good user experience.

- To help in debugging, log the version of the framework that your driver is using. Both KMDF and UMDF provide functions that return the version to which the driver is bound.
- Check the WHDC Web site and the driver discussion forums at Microsoft and Open Systems Resources (OSR) for information about new releases.

Resources

Windows Driver Foundation (WDF)on the WHDC Web site:
http://www.microsoft.com/whdc/driver/wdf/default.mspx

White Papers:

Introduction to the Windows Driver Foundation
http://www.microsoft.com/whdc/driver/wdf/wdf-intro.mspx

Architecture of the Windows Driver Foundation
http://www.microsoft.com/whdc/driver/wdf/wdf-arch.mspx

Architecture of the Kernel-Mode Driver Framework
http://www.microsoft.com/whdc/driver/wdf/kmdf-arch.mspx

Architecture of the User-Mode Driver Framework
http://www.microsoft.com/whdc/driver/wdf/umdf-arch.mspx

Driver Tips:

Troubleshooting KMDF Driver Installation
http://www.microsoft.com/whdc/driver/tips/KMDF_install.mspx

WDK Documentation:

Kernel-Mode Driver Framework
"Framework Library Versions"
http://msdn.microsoft.com/library/en-us/kmdf_d/hh/KMDF_d/ch0_dfarchoverview_7682c339-92b6-4b94-9fc5-3226e4318a65.xml.asp

User-Mode Driver Framework
"IWDFDriver"
http://msdn.microsoft.com/library/en-us/UMDF_r/hh/UMDF_r/umdfobjectref_2bce205e-d670-4dae-870a-f5b01c3ea49e.xml.asp?frame=true

© 2006 Microsoft Corporation. All rights reserved.